گزارش کار تکلیف شماره ۳

در این تکلیف، با استفاده از کتابخانه Lucene، سعی در انجام نمایه‌گذاری بر روی متون هستیم.

چالش: با توجه به وجود برخی امکانات موردنیاز که به طور کامل در نسخه ۸ Lucene موجود نبود، از نسخه ۷ این کتابخانه استفاده شده است.

- **گروه ۳:**
  - دینا امیدوار طهرانی
  - پرستو باقرپور
  - نسیم فانی
  - محسن محمودزاده
  - مریم واقعی

۱. خواندن ورودی

در ابتدای امر، آدرس دایرکتوری فایل داده ورودی(dataDir) و آدرسی که نمایه‌ها قرار است در آن نوشته شوند(indexDir) را از کاربر دریافت می‌کنیم. سپس، با استفاده از تابع readFile کتابخانه File جاوا، محتویات فایل ورودی را در متغیری به نام content ذخیره می‌کنیم.

```
BufferedReader reader =
        new BufferedReader(new InputStreamReader(System.in));
System.out.print("please Enter index directory path:: ");
indexDir = reader.readLine();
System.out.print("please Enter text file path for indexing:: ");
dataDir = reader.readLine();
content = readFile(dataDir, StandardCharsets.UTF_8);
```

۲. پیش‌پردازش ورودی و نمایه گذاری

- analyzer

برای پیش‌پردازش، باید analyzer را برای توکن‌سازی از محتوای ورودی و آماده‌سازی به منظور نمایه‌گذاری، بسازیم. می‌توان از analyzer های متخلفی جهت انجام این مهم استفاده کرد(StandardAnalyzer). ما در این جا از CustomAnalyzer که از کلاس انتزاعی Analyzer ارث‌بری می‌کند، استفاده می‌نماییم. با اضافه‌کردن آپشن‌ها مطابق نیاز(ریشه‌یابی، حذف کلمات توقف، کوچک‌سازی حروف و ...)، analyzer مورد نظر را می‌سازیم.

```
Analyzer analyzer = CustomAnalyzer.builder()
        .withTokenizer("standard")
        .addTokenFilter("lowercase")
        .addTokenFilter("stop")
        .addTokenFilter("porterstem")

        .build();
```

- getDocuments

در ادامه باید از محتوای فایل ورودی یعنی content، اسناد را به دست آوریم. این کار را با تابع getDocuments صورت می‌گیرد. در این تابع، فایل خط به خط خوانده شده و از خطوطی که محتوای آنها جزئی از اطلاعات مورد نظر جهت نمایه گذاری نیست، صرف نظر می‌شود. با درنظر گرفتن شرط i != 0 ، هنگامی که به چنین خطوطی می‌رسیم یعنی در پایان سند هستیم. بنابراین باید اطلاعات هر خط را که به کمک متغیر str ذخیره کرده بودیم را در قالب یک سند ذخیره کنیم. ابتدا متن آنالیز شده را به کمک تابع getAnalyzedString به دست می‌آوریم. این تابع در ادامه به طور کامل توضیح داده خواهد شد. با داشتن رشته stringAnalyzed که نسخه پیش پردازش شده محتوای ورودی ماست، می‌توانیم سند را بسازیم. از آنجا که در لوسین، یک سند از چند Field تشکیل می‌شود، به سند doc، یک نمونه Field از نوع Text یا TextField اضافه می‌کنیم. در این نمونه TextField، نام فیلدی که TextField باید از روی آن ساخته شود و همینطور مقدار فیلد را به عنوان آرگومان ورودی به سازنده TextField می‌دهیم. همچنین چون در آینده به مقدار فیلد [که در رشته stringAnalyzed ذخیره شده است] نیاز داریم، آن را با استفاده از کانفیگ Field.Store.YES ذخیره می‌کنیم. در نهایت سند doc را به لیست تمامی اسناد (docs) اضافه می‌نماییم. در نهایت سند پایانی را هم پیش‌پردازش کرده و به لیست تمامی اسناد اضافه می‌کنیم.

```
Scanner scan = new Scanner(content);
ArrayList<Document> docs = new ArrayList<>();
int i=0;
String str = "";
while (scan.hasNextLine()){
    String readLine = scan.nextLine();
    if(readLine.contains(".I "+(i+207))){
        if(i!=0) {
            String stringAnalyzed = getAnalyzedString(analyzer,str);
            Document doc = new Document();
            doc.add(new TextField("content",stringAnalyzed,Field.Store.YES));
            docs.add(doc);
        }
        str = "";
        i++;
    }else if(!readLine.contains("I "+(i+207)) && !readLine.contains(".W")){
        str+=" "+readLine;
    }
}
//last paragraph remain so it must be documented
Document doc = new Document();
String stringAnalyzed = getAnalyzedString(analyzer,str);
doc.add(new TextField("content",stringAnalyzed,Field.Store.YES));
docs.add(doc);
return docs;
```

- analyze

در ادامه قصد داریم تا IndexWriter را بسازیم. ابتدا به کمک FSDirectory، دایرکتوری ای که قصد نوشتن نمایه‌ها در آن را داریم را باز می‌کنیم.

ابتدا با استفاده از IndexWriterConfig، پیکربندی موردنظر خود را برای ساختن IndexWriter مشخص می‌کنیم. analyzer سفارشی ساخته شده در مرحله قبل بخشی از این پیکربندی است. سپس مد باز یا OpenMode را برای IndexWriter بر روی حالت CREATE قرار می‌دهیم. در این حالت، یا یک نمایه جدید ساخته می‌شود و یا بر روی نمایه موجود بازنویسی صورت می‌گیرد.

حال با پیکربندی موجود و دایرکتوری نمایه‌ها، یک نمونه از IndexWriter را با نام writer می‌سازیم. با پیمایش بر روی اسناد، سند ها را به نمایه اضافه می‌کنیم. جهت جلوگیری از بازنویسی بر روی سند اولیه، پس از نمایه‌گذاری سند اول، OpenMode را برای IndexWriter به حالت CREATE_OR_APPEND تغییر می‌دهیم. در این حالت، اگر نمایه موجود باشد، اسناد آینده به سند ضمیمه(append) می‌شوند. پس از نمایه‌گذاری تمامی اسناد، writer را می‌بندیم.

```
Directory dir = FSDirectory.open(Paths.get(indexDir));

IndexWriterConfig iwc = new IndexWriterConfig(analyzer);
iwc.setOpenMode(IndexWriterConfig.OpenMode.CREATE);
IndexWriter writer = new IndexWriter(dir, iwc);
for(int i=0 ; i<documents.size() ; i++){
    writer.addDocument(documents.get(i));
    if(i==0) iwc.setOpenMode(IndexWriterConfig.OpenMode.CREATE_OR_APPEND);
}
writer.close();
```

- getAnalyzedString

برای نمایه‌گذاری، باید توکن ها و در ادامه، term ها را به دست آوریم. پس به کمک یک analyzer، ابتدا با متد tokenStream یک توالی از توکن‌ها را به دست می‌آوریم. این متد، نام فیلدی که این tokenStream برای آن ساخته شده، و reader ای را که جریان توکن ها از آن خوانده می‌شود، به عنوان ورودی گرفته و یک TokenStream برمی گرداند. این نمونه را tokenStream می‌نامیم.

حال با استفاده از کلاس های پیش فرض در لوسین و جاوا، خصیصه‌هایی برای توکن ها ایجاد کرده تا در پردازش‌های پیشرو از آنها استفاده کنیم. از آنجا که کلاس TokenStream از کلاس AttributeSource ارثبری می‌کند، می‌توانیم متد addAttribute والد را فراخوانی کنیم. هر نمونه از کلاس AttributeSource شامل لیستی از AttributeImpl ها و متدهایی برای اضافه‌کردن یا گرفتن آنهاست. تنها یک نمونه از یک خصیصه در یک AttributeSource معین وجود دارد(مطابق الگوی طراحی singleton). برای اطمینان از این موضوع، نوع Attribute را به متد addAttribute(Class) پاس می‌دهیم؛ این متد بررسی می‌کند که تاکنون نمونه‌ای از این نوع کلاس وجود دارد یا خیر. در صورت وجود آن را برمی‌گرداند و در غیر این صورت آن را ایجاد کرده و سپس برمی‌گرداند. به همین ترتیب دو خصیصه از نوع CharTermAttribute (برای به دست آوردن termها) و OffsetAttribute(برای به دست آوردن موقعیت کاراکترهای توکن) برای tokenStream می‌سازیم.

حال باید توکن‌ها را به دست آوریم. با فراخوانی متد incrementToken روی tokenStream، یکی یکی به توکن ها دسترسی پیدا می‌کنیم. نکته قابل توجه این است که پیش از استفاده از این متد، باید از متد reset استفاده کرد. سپس آدرس‌های شروع و پایان توکن جاری را به دست آورده و در نهایت term را به دست می‌آوریم. برای جستجوی termها در بین اسناد، term ها را در یک مجموعه ذخیره می‌نماییم. همچنین term ها را با استفاده از WhiteSpace در یک رشته(تحت نام stringAnalyzed) نگه می‌داریم. در نهایت پس از پیمایش و پردازش تمامی توکن‌ها، tokenStream را می‌بندیم. حاصل اجرای این تابع، رشته آنالیزشده است.

```
TokenStream tokenStream = analyzer.tokenStream("content",content);
OffsetAttribute offsetAttribute = tokenStream.addAttribute(OffsetAttribute.class);
CharTermAttribute charTermAttribute = tokenStream.addAttribute(CharTermAttribute.class);

String stringAnalyzed = "";
try{
    tokenStream.reset();
    while (tokenStream.incrementToken()) {
        int startOffset = offsetAttribute.startOffset();
        int endOffset = offsetAttribute.endOffset();
        String term = charTermAttribute.toString();
        hash_Set.add(term);
                    System.out.println(term);
        stringAnalyzed+=term+" ";
    }
    tokenStream.close();
}catch (IOException e){
    System.out.println(" caught a " + e.getClass() + "\n with message: " + e.getMessage());
}
return stringAnalyzed;
```

نکته قابل توجه دیگر اینست که به علت وجود برخی خطاهای منطقی در فرایند آنالیز داده ها[که بیشتر به analyzer های موجود در کتابخانه لوسین مربوط می‌شود]، آنالایزر مورد استفاده در متد getDocuments، CustomAnalyzer و آنالایزر مورد استفاده در متد analyze، آنالایزرهای SimlpeAnalyzer یا WhitespaceAnalyzer هستند. استفاده از StandardAnalyzer [که در فرایند توسعه این نمایه‌گذار مورد استفاده و آزمون قرار گرفت] به علت حذف کامل کلمات توقف و عدم هماهنگی با دیگر گام‌های پیش پردازش توصیه نمی‌شود(مثلا کلمه one توسط ریشه‌یاب به on تقلیل پیدا می‌کند و حال StandardAnalyzer این توکن را به اشتباه کلمه توقف تشخصیص داده و آن را حذف می‌نماید).

از دیگر چالش‌های پیش‌آمده، فرایند توکن‌سازی کلمات ورودی بود که در این مسیر، آنالایزرهای مختلفی(مانند EnglishAnalyzer، SnowballAnalyzer، EnglishStemmer، StopFilter، PorterStemFiltetr، CJKAnalyzer، LeafReader و ...) مورد آزمون و خطا قرار گرفت که به دلایل گوناگون عدم پشتیبانی در نسخه‌های متفاوت، عدم ارائه نتایج مطلوب و ... مورد استفاده قرار نگرفتند.

```
Analyzer analyzer = CustomAnalyzer.builder()
        .withTokenizer("standard")
        .addTokenFilter("lowercase")
        .addTokenFilter("stop")
        .addTokenFilter("porterstem")

        .build();

ArrayList<Document> docs = getDocuments(content,analyzer);

Analyzer analayzer2 = new SimpleAnalyzer();
analyze(docs,analayzer2);
```

۳. جستجوی کلمات اسناد

فرایند جستجوی کلمات اسناد را به کمک کلاس Searcher انجام می‌دهیم. در این کلاس ابتدا فایل های لازم برای خواندن نمایه‌ها(دایرکتوری index) و نوشتن نتایج جستجوی کلمات در اسناد(فایل result.txt) را می‌سازیم.

```
File resultFile = new File("result.txt");
FileWriter writer = new FileWriter("result.txt");
Directory indexDirectory = FSDirectory.open(Paths.get(indexDirectoryPath));
IndexReader reader = DirectoryReader.open(indexDirectory);
```

در ادامه برای جستجو در نمایه‌ها، یک نمونه از کلاس‌های IndexSearcher و QueryParser می‌سازیم. IndexSearcher همانطور که از نام آن پیداست برای جستجوی نمایه‌ها مورد استفاده قرار می‌گیرد. با استفاده از queryParser نیز می‌توان یک پرس‌وجو را به searcher داد تا نتایج پرس‌وجو را برگرداند.

```
IndexSearcher searcher = new IndexSearcher(reader);
queryParser = new QueryParser("content",analyzer);
```

برای یافتن تعداد تکرار هر term در هر سند به طور مجزا، ابتدا اسنادی که شامل term هستند را می‌یابیم و سپس در داخل هر یک از این اسناد، تعداد رخدادهای term را به دست می‌آوریم . با استفاده از نمونه HashSet مقداردهی‌شده در متد getAnalyzedString، از تمامی term های پیش‌پردازش‌شده درون اسناد، یک به یک، پرس‌وجو ساخته و با searcher در نمایه‌ها جستجو می‌کنیم. این جستجو به کمک متد search صورت می‌گیرد. متد search در اسناد، مطابقت‌های(hit) متن پیمایش شده با پرس‌وجو را برمی‌گرداند. از این رو آرگومان دوم ورودی(n) از ما می‌خواهد تا مشخص کنیم که خواستار چه تعداد از مطابقت‌های اولیه هستیم. با مقداردهی این آرگومان با Integer.MAX_VALUE اطمینان می‌یابیم که تمامی مطابقت‌ها توسط queryParser و searcher در اختیار ما قرار می‌گیرد. خروجی متد search، یک نمونه از کلاس TopDocs است که مطابقت‌ها را در خود ذخیره می‌کند. با استفاده از خصیصه‌ی scoreDocs این نمونه، می‌توانیم امتیاز رتبه‌بندی ter اسناد را به دست آوریم. لیست تمامی این امتیازات به ازای هر پرس‌وجو، در آرایه sd نگه‌داری می‌شود. حال به ازای هر امتیاز، به کمک IndexReader، فیلدهای سندی که این امتیاز را دارد، در نمایه پیدا می‌کنیم. برای پیدا کردن این سند، از فیلد doc نمونه ScoreDoc می‌توان استفاده کرد. حال به مقدار هر فیلد از سند، به کمک متد get دسترسی پیدا می‌کنیم. برای به دست آوردن تعداد تکرار هر term در هر سند، متد countOccurences را فراخوانی می‌کنیم. تا به اینجا ما به ازای یک term معین در نمونه HashSet، تعداد رخداد آن را در هر سندی که در آن وجود دارد، به تفکیک به دست آورده‌ایم. این فرایند برای تمامیtermهای درون نمونه HashSet انجام می‌شود. نتایج در فایل result.txt نوشته و نگه‌داری می‌شوند.

```
for(String queryString : hash_set){
    Query q=queryParser.parse(queryString);
    TopDocs td = searcher.search(q,Integer.MAX_VALUE);
    ScoreDoc[] sd = td.scoreDocs;
      System.out.print(queryString+": ");
    writer.append(queryString + ": ");
    for (int j = 0 ; j< sd.length ; j++){
        Document document = reader.document(sd[j].doc);
        String fieldContent = document.get("content");
          System.out.println(fieldContent);
        int count = countOccurences(fieldContent,queryString);
          System.out.print("doc"+sd[j].doc+"["+count+"]");
        writer.append("doc" + sd[j].doc + "[" + count + "]");
        if(j < sd.length-1)  {
            writer.append(", ");
              System.out.print(", ");
        }
    }
    writer.append("\n");
      System.out.println();
}
writer.close();
```

```
static int countOccurences(String str, String word)
{
      // split the string by spaces in a
    String a[] = str.split(" ");

      // search for pattern in a
    int count = 0;
    for (int i = 0; i < a.length; i++)
    {
        // if match found increase count
        if (word.equals(a[i]))
            count++;
    }
    return count;
}
```

نمونه ساخته شده از کلاس Searcher در تابع main مورد استفاده قرار می‌گیرد. از آنجا که فرایند توضیح داده شده در بالا، در سازنده‌ی این کلاس انجام می‌شود، با نمونه‌سازی از کلاس Searcher، نتایج را در فایل result.txt خواهیم داشت.

```
Searcher s = new Searcher(indexDir,analayzer2,hash_Set);
```

لازم به ذکر است که در فاز نمایه‌گذاری و جستجوی کلمات اسناد، چالش های گوناگونی پیش رو قرار گرفت که مهم‌ترین آن، به دست آوردن تعداد تکرار یک term در هر سند به طور مجزا بود که برای وصول به چنین هدفی روش‌های گوناگونی چون کار با TopDocs، tfidfSim، PostingsEnum، docFreq، TermVector و .... مورد بررسی قرار گرفتند.

کلمات اسناد به همراه تعداد رخدادشان در هر سند، در فایل result.txt نوشته شده است.