White Paper: Towards Developing A Comprehensive API For the Kurdish Language

Abdul-Rahma Mawlood-Yunis
Physics and Computer Science Department, Wilfrid Laurier University, Waterloo, Canada

Abstract

In this work, we present our effort towards building a comprehensive application programming interface (API) for processing and mining text written in the Kurdish language, a language with similar scripts to Arabic and Farsi. To the best of our knowledge, this effort is the first of its kind. The current version of the API supports essential text processing and can be used to develop several language applications. We present our experience dealing with text processing challenges. We also present an efficient algorithm for processing and sorting a large number of text files. This work has three main contributions: First, a preliminary API for processing the Kurdish language, i.e., the code library. Second, an algorithm for processing and sorting large data sets on non-powerful user devices. Third, a large data set that has been collected from online news articles, blogs, and review documents. The size of the dataset is about 12 megabytes and includes over 500,000 unique words.

1 Introduction

Most of the success achieved in NLP is for popular languages, like English, or other languages that have text corpora of hundreds of millions of words. These languages, however, are only about 23 languages from more 7,000 languages spoken in the world[1]. The majority of human languages are in dire need of tools and resources to overcome the resource barrier so that NLP can deliver enormous widespread benefits (datafloq, 2020). These languages, be they minority, regional, endangered, or heritage languages, are called low-resource languages, languages that lack the sufficient linguistic resources needed for building statistical NLP applications. Expanding the range of languages traditionally studied by NLP, i.e., studying NLP for low-resource languages, can be utilized as a testbed for validating and advancing the current NLP methods and techniques. For example, new developments in low-resource NLP give insight into the connection between the word and the meaning, not by pure statistics, but by comparing diverse languages (Ruder, S., et al 2019).

In this work, we present our efforts to study one of the low resource languages, i.e., the Kurdish language. The Kurdish language is spoken by some 40 million[2] people in Turkey, Iraq, Iran, Syria, and other parts of the world. We are focusing on building a comprehensive application programming interface (API) for processing and mining text written in the Kurdish language, a language with similar character glyphs to Arabic and Farsi. The current version of the API supports essential text processing and can be used to develop several language applications and games. The current state of the API supports file management, text segmentation, and statistical analysis of text among other operations. It will play a vital role in building several immediate applications including a Kurdish spell checker, a Kurdish thesaurus, automatic translation, converting text to speech, and information retrieval applications. We also present an efficient algorithm that we have developed in the course of this study for processing and sorting large

[1] https://www.ethnologue.com/guides/how-many-languages
[2] https://www.britannica.com/topic/Kurdish-language

data sets on non-powerful user devices, for example, a developer's laptop. The API possesses a unique characteristic; its use is not limited to the Kurdish language. It can be used for processing text in other languages that can be represented in the UTF-8 encoding. These include languages such as Arabic, Farsi, Pashto, and Urdu.
This work has three main contributions: First, a preliminary API, or code library, for processing the Kurdish language. The API is written in the Java programming language and can be added as an external package to any Java project for further development. Second, an algorithm for processing and sorting large data sets on non-powerful user devices. Third, a large dataset that has been collected from online news articles, blogs, and review documents. The size of the dataset is about 12mb and includes 533,442 unique words. The data set generated from this work will add to the wealth of datasets available by processing online texts and can be utilized as a testbed for validating and advancing recent approaches to modeling multilingual sentences and multilingual word embeddings.

The rest of this work is organized as follows: In Section two, some important related works are reviewed. In section three, several Kurdish text processing issues and challenges are discussed. In section four, implementations of key features of the current Kurdish API (KAPI) are discussed. In section five, instructions on how to set up and configure an integrated development environment (IDE) for working with the KAPI is provided. In section six, an example of a KAPI application is presented and some future applications have been pointed out. In section seven, an algorithm for processing a large data set is presented and discussed. And finally, in section eight, the paper is concluded.

2 Related Work

Collecting data from online texts for low resource languages and providing statistics about each of these languages, such as word frequencies and contexts, requires extensive effort; however, it is proven to yield useful results. One of the most notable examples of such work is the Crubadan project[3] (K. Scannell, 2003). Other examples include the Human Language Program (Abney, and Bird, 2010) and the Leipzig corpora collection[4](C. Biemann, 2007). In each of these projects, a large number of datasets for a large number of languages are created form processing online texts.
Different from collecting data online and creating language models and corpora for individual languages, more recent approaches explore the commonalities among languages to build new language models (Barret Zoph, et.al. 2019, Toan Q. Nguyen, David Chiang, 2017, Steven Abney & Steven Bird, 2019). The new approaches use the morphological, lexical, and syntactic similarities among languages to create a new language model for a particular language from an existing model and apply obtained results from one language to another. For example, they train a model in English as a resource-rich language and generate an out-of-the-box model in a low-resource domain. The language model transfer is done at an annotation level such as Part Of Speech (POS) tags and syntactic or semantic features via cross-lingual bridges (e.g., word or phrase alignments).
The latest recent efforts to model languages trained using 100 languages and cross-lingual sentence embeddings. The latter approach learns joint multilingual sentence representations for 93 languages belonging to more than 30 different language families and written in 28 scripts. They convert data in multiple languages to a shared representation (e.g., multilingual word vectors) and train a single model on a mix of datasets of languages to build large and more focused multilingual processing systems. (Mikel Artetxe, Holger Schwenk, 2019). Another ongoing work is Universal Dependencies[5] (UD) framework for creating treebanks. It is an open community framework for consistent annotation of grammar across different human languages. It has produced more than 150 treebanks in 90 languages and aims to support comparative evaluation and cross-lingual learning and facilitate multilingual natural language processing and enable comparative linguistic studies.
The data set generated from this work will add to the wealth of datasets available by processing online texts, the first approach. It also can be utilized as a testbed for recent approaches to model multilingual sentences and multilingual word embeddings, i.e., the second approach, or the more recent approaches.

[3] http://crubadan.org/applications
[4] https://wortschatz.uni-leipzig.de/en
[5] https://universaldependencies.org/

3 NLP for the Kurdish Language

In this section, we discuss some essential issues that need to be understood when processing text in Kurdish and other similar languages. Proper understanding of these issues is vital for successful application implementation in Kurdish and other related languages

3.1 Word Size

Kurdish words have a clear boundary. In other words, Kurdish words are separated from each other by a space. This characteristic is a very useful feature for Kurdish text processing. However, determining the size of Kurdish words is less obvious.

The issue here is that Kurdish words could contain a ZERO WIDTH NON-JOINER character (ZWNJc). This happens because the Kurdish language is written in cursive. The ZWNJc is used to separate two letters from joining to each other. Examples of such words include Newly and Burger (کفتەی، نوێی). Every time a letter 'y' is added to a word which ends with either another 'y' or an 'h' (in Kurdish 'ی' or 'ێ' and 'ە'), we will encounter this issue. Not counting for ZWNJc results in an incorrect determination of word length, and thus, operations depend on it. For example, using any existing programming language, a function call to determine the size of words such as Grapy or Burger (ترێێ or کفتەی) will return five and six respectively while it should be four and five.

A simple solution for this issue would be to write a simple function for calculating word length in which a ZWNJc has not been counted as a letter. The following code snippet shows how we overcome this issue in the current version of the Kurdish API.

In **ALGORITHM 1**, the input is a word which we would like to determine its length, and the output would be "wordLength", a variable which holds word's length.

ALGORITHM 1. Word Length

Input: String Word
Output: The length of Word after without counting ZWNJc

```
int wordLength = word.length() ;
char[] temp = word.toCharArray() ;
        for (int i=0; i< temp.length ; i++){
         int c = (int)temp[i] ;
                if (c==8204) { // 8204 is the int value of ZWNJc
                  wordLength -- ;
        } // end if
      }// end for
```

3.2 Digraph in the Kurdish Lanague

Similar to the English language where sometimes two letters make one sound, e.g. sh, ch, th, ph, etc., in the Kurdish language, both the letters L (ل) and strong L (LL) can be combined with the letter A (ا) to make one sound, 'لا' and 'ڵا'. Examples of words which include such letters are but (بەڵام), answer (وەڵام), and village (لادێ) among thousands. Also, similar to English phonic sounds, wo, oo, and ai, the Kurdish language uses two w letters (وو) in words such as two (دوو), done (بوو), and hair (موو). Business analysts and application developers need to be aware of these issues to implement their applications correctly. For example, while it is acceptable to consider the size of the word two ('دوو') to be three letters, a decision needs to be made on the size of the word but ('بەڵام'). It could be the size of four, five (when considering 'ڵا' to be made of the letters L and A), or six (when considering 'ڵا' to be made of the letters L and A and accents on the letter L) characters.

3.3 Letter Representation Discrepancy

In UTF-8 encoding [The Unicode Standard, 2013], integer representations of Kurdish characters are not in order. For example, while the letter p (پ) is the third letter in the Kurdish alphabet, its integer value is not. This is because the Kurdish language uses a subset of the Arabic alphabet and ten additional Kurdish scripts (پ، چ ، ڕ، ژ ، ڤ، گ ، ڵ ، ۆ ، ێ) that do not exist in the Arabic alphabet. For example, if you execute this piece of code: print (((int) 'b' - (int) 'c')) you will receive -1. The result indicates that

the letter b precedes the letter c, and the distance between the two letters is one. However, if you execute the same piece of code using Kurdish letters, you will get different results; the code print ((int) 'ب' - (int) 'پ')) will return -86.
The discrepancy in Kurdish letter representation, word size, and digraph issues, prevents using STRING, i.e., text, operations incorporated in high-level programming languages. Thus, operations such as sorting and searching algorithms and applications that rely on STRING comparison will be incorrect. To handle these issues, STRING operations have been overridden in the KAPI.

4 Current Kurdish API

In this section, we discuss some of the functionalities that have been implemented in the current Kurdish API.

4.1 Reading and Writing Text

What is unique about reading Kurdish text files is the use of "UTF-8" parameters with the Java input/output readers. Further, InputStreamReader and OutputStreamWriter classes need to be used for reading and writing text files respectfully. The following code snippet shows what is needed to read and write a text file written in UTF-8 encoding. The code consists of three main segments: creating reader/writer object (steps from1 to 4), processing reading/writing text (step 5), and closing the input/output files (step 6).

/*create reader/writer object*/

1. Reader reader = new InputStreamReader(new FileInputStream(
"file-directory\\inputFileName.txt"), "UTF-8")
2. BufferedReader fin = new BufferedReader(reader)
3. Writer writer = new OutputStreamWriter(new FileOutputStream(
"file-directory\\outputFileName.txt"), "UTF-8")
4. BufferedWriter fout = new BufferedWriter(writer)

/* processing reading/writing text*/

5. while ((s = fin.read()) != -1) {
fout.write((char)s)
}
/*close the input/output files*/

6. fin.close(); fout.close() ;

4.2 Word Count

Currently, for its usefulness, most text editors provide the word count functionality. This functionality is also valuable in the context of application development and text processing. That is, other application functionalities could make use of or depend on the word count. The current Kurdish API provides the word count function for input text. Further, word count can also be used with other functionalities. Examples of such functionalities include retrieving words that start or end with a specific letter. For example, using the current version of the KAPI, we can count how many words in the document start/end with the letter ‘d’.

4.3 Word Specific Procedures

The current KAPI provides various text processing methods. These include word specific routines such as extracting words that start with specific letters, extracting words that end with specific letters, extracting words of a specific length (e.g., two, three, four, etc.), extracting all special characters (e.g. %, #, &, @, etc.), extracting the most used words, sorting extracted words, etc. These routines and others are very useful to perform statistical analysis on text.

5 Groundwork For Creating Kurdish API

Using the Kurdish language with an integrated development environment (IDE) requires system configuration. In this section, we describe Eclipse IDE configuration for the Kurdish language and Java programming language for the windows system. This description helps the future developer to set up their working environment correctly. Further, the configuration is not limited to the Kurdish language; other languages that utilize UTF-8 character encoding (for example, Arabic, Persian, and Urdu) can be configured in the same way.

5.1 New System Variable

To be able to compile code where UTF-8 characters are used for string initialization and/or comments in Eclipse, a new system variable needs to be declared. The variable name should be JAVA_TOOL_OPTIONS and its value should be -Dfile.encoding=UTF8. This can be achieved following these steps:

On the windows control panel, click on the following tabs in sequence: System and Security → System →advance system settings → Environment variable → create new user variable. Name the variable JAVA_TOOL_OPTIONS and put inside the variable the value -Dfile.encoding=UTF8.

5.2 Setting the Font Type

To run and see the Kurdish language (Arabic, Farsi, and Urdu as well) in the Eclipse console, one of the following two steps, or both, need to be done:

1. The UTF-8 coding option needs to be selected from the common tab on the run configuration. This can be achieved following this sequence of clicks: inside Eclipse click on run → Run configuration → common → select UTF-8 coding.
2. The console font needs to be changed to a font type that supports UTF-8 encoding. An example of a font type that supports UTF-8 encoding would be the Tahoma font. This step can be achieved as follows: inside Eclipse click on Window → Preferences → General → Appearance → Colors and Fonts → Debug → Console and edit the font.

5.3 Virtual Machine Parameters

Where classes and methods' comments are written in the Kurdish language, the UTF-8 encoding parameter needs to be passed to the Java Virtual machine to generate Java documentation using the Javadoc program. This step can be achieved by clicking on the following tabs in sequence: inside Eclipse click on project tab → generate Javadoc → next → next → in entry filed for extra vm options type "-encoding UTF-8 -charset UTF-8 -docencoding UTF-8".

6 An Algorithm For Processing Large NLP Data Sets

Most natural language processing (NLP) applications deal with large data sets. Hence, the efficient processing of such big data sets is an important step towards developing any valuable NLP application. In this section, we present a simple yet powerful algorithm that we have developed for sorting large data sets on non-powerful user devices, for example, the developer's laptop.

The algorithm can be positioned under a class of algorithms knows as a streaming algorithm [Anna C. Gilbert, et al, 2001]. It is a 2-pass algorithm which means that the algorithm goes back to the processed input for additional processing. The algorithm is made of two main steps; it converts the large data sets into a list of small size sorted files, and merges the sorted files into one large sorted file. Below, we describe these two steps, and the pseudocodes for the two steps are provided in the appendix of this article.
We used this algorithm to process a large number of files collected from the web and create a large dataset that can be provided to the researcher upon request.

6.1 Converting Large Data Sets Into a List of Small-Size Files

Collecting text from the web for the Kurdish language can result in hundreds of thousands or millions of small size text files with hundreds of megabytes or gigabytes of data. Processing all this text in the virtual memory of small devices, such as a conventional (8-16 GB RAM) desktop or laptop, is difficult. The processing is cumbersome either because of out of the virtual memory problem or slow data processing. To overcome the slow data processing and/or out of the virtual memory problem, we need to keep and process only a small portion of data in computer memory at any instance of time. We overcame the described problem by converting the large input data to a list of significantly smaller files. The algorithm's main steps are one, reading input data, two, converting input data into a manageable number of files, three, simultaneously sorting these files, and four, writing back the intermediate result into the user device.
Converting the input dataset into a manageable number of files depends on how the input data is stored or organized. For example, when the input data consists of a very large number of small size files, we merge thousands of files into a single file (every 3000 to 5000 files into one new file). The number of files to be merged into one file depends on the size of the small files. That is, the smaller the size of the file, the large the number of files to be merged into one file is and vise versa. However, if the input data is made of one giant file, we divide this file into a manageable number of files with each file holding a fraction or percentage of the original data.

The sorting step is achieved using the TreeSet data structure. That is, during the file reading, words are added to the TreeSet data structure in which duplicated words are removed and sorted in descending order. The TreeSet data structure is implemented in most high programming languages, hence, we did not re-write a sorting routine. However, if interest arises, calls to add words to the TreeSet data structure can be replaced with locally developed sorting algorithms and data structures. Algorithm 4, is the pseudocode for the described steps.

6.2 Merge Sorted Files

In this part of the algorithm, the short-list of files created in the previous step is used to produce the end-result sorted file. The algorithm starts by creating a new file from merging the first two files from the short-list and saving the newly created file on the user's device to be used in the next step. In the next step, the third file from the short-list is merged and sorted with the file just created in the previous step to create yet another output file which in turn merges with the fourth file from the list. These steps are repeated until the last file from the short-list is merged with the file created in the immediate prior step. It is important to note that at each step of the merging procedure, only a portion of the data is processed which in turn helps less powerful devices, such as conventional (8 GB RAM) laptops, to operate on such large input data and memory-intensive procedures. Further, after each step, the unwanted intermediate data results get deleted. This is to avoid extensive usage of storage. The pseudocode for this step is described in Algorithm 5 which is provided in the appendix of this article.

7 Sample Use of KAPI and Its Future Applications

In this section, we provide an example where we used our current KAPI for processing Kurdish text and generate the diagram. We also name some future applications that need KAPI as a prerequisite.

7.1 Most Frequent Words in A School Text Book

We have used the current KAPI to study some aspects of the prime textbook which is used for teaching grade one elementary school students in the Kurdistan region of Iraq for the year 2020.

The aspects include extracting all words in the book, determining the most frequently used words and generating the word frequency graph, calculating the total number of unique words used in the book (which is 994 words), and extracting and counting all words that start with similar letters. Figure 1 shows some of these statistics. The x-axes of the graph represent the 15 most frequent words, and the y-axes represent the frequency of each word. The example demonstrates that the KAPI can be used promptly for a large number of studies and applications that otherwise would be impossible or difficult to achieve.

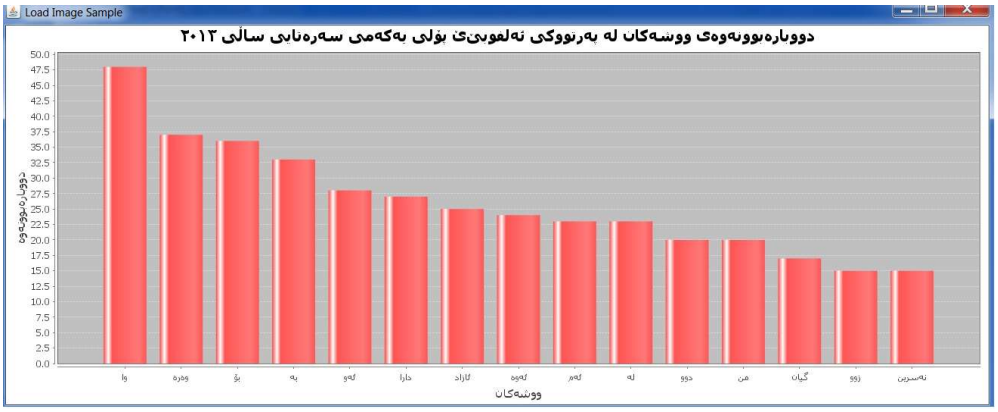

Figure 1, the most frequently used words in the prime grade one textbook in Kurdistan

7.2 Description of Future Applications

The KAPI is the base for building a large number of viable applications. The applications include a Kurdish spell checker, a Kurdish thesaurus, a Kurdish word bank, identifying common words among Kurdish dialects, and finding the frequently used Kurdish words. These applications are some of the future works that we would like to explore using the KAPI.

8 Conclusion

In this work, we introduced the initial version of the Kurdish application programming interface for text processing and mining. We discussed some of the operations and applications that the current KAPI supports. We also presented an algorithm for processing and sorting large NLP data sets that we developed during this study. We explained that the use and benefits of the current KAPI go far behind the Kurdish language; it can be used for processing text written in languages such as Arabic, Farsi, and Urdu as well. We provided the necessary information to the application developers and programmers to set up their IDE and discussed some immediate applications as well as future research directions.

9 References

Anna C. Gilbert, Yannis Kotidis, S. Muthukrishnan, and Martin Strauss. 2001. Surfing Wavelets on Streams: One-Pass Summaries for Approximate Aggregate Queries. In Proceedings of the 27th International Conference on Very Large Data Bases (VLDB '01), Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard Thomas Snodgrass (Eds.). *Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 79-88.*

Barret Zoph, Deniz Yuret, Jonathan May, Kevin Knight. 2019. Transfer Learning for Low-Resource Neural Machine Translation. In Proceedings of the 2016 Conference on Empirical Methods in *Natural Language Processing, pages 1568–1575.*

Biemann, Chris; Heyer, Gerhard; Quasthoff, Uwe and Richter, Matthias. 2007. The Leipzig Corpora Collection – Monolingual corpora of standard size. In *Proceedings of Corpus Linguistics 2007, Birmingham, UK, 2007.*

Chao Jiang, Hsiang-Fu Yu, Cho-Jui Hsieh, and Kai-Wei Chang. 2018. Learning Word Embeddings for Low-Resource Languages by PU Learning. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1, 1024–1034.*

K. Scannell. 2003. Automatic thesaurus generation for minority languages: an Irish example. *In Actes de la 10e conférence TALNa Batz-sur-Mer, vol. 2, pp. 203–212.*

Karl Moritz Hermann, Phil Blunsom. 2014. Multilingual Models for Compositional Distributed Semantics. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1), pages 58–68.*

Mikel Artetxe, Holger Schwenk. 2019. Massively Multilingual Sentence Embeddings for Zero-Shot Cross-Lingual Transfer and Beyond. *Journal of Trans. Assoc. Comput. Linguistics (7), 597-610.*

NLP for Low-Resource Settings, [online, retrieved August 24, 2020] Available: *https://datafloq.com/read/nlp-for-low-resource-settings/7019.*

Ruder, S., Vulić, I. and Søgaard, A. 2019. Survey of Cross-lingual Word Embedding Models. *Journal of Artificial Intelligence Research, pages 569–631.*

Steven Abney & Steven Bird. 2010. The human language project: Building a universal corpus of the world's languages. In *Proceedings of the 48th annual meeting of the association for computational linguistics (pp. 88-97). Uppsala: Association for Computational Linguistics.*

The Unicode Standard Version 6.2 – Core Specification, web-site available at *http://www.unicode.org/versions/Unicode6.3.0/*

Toan Q. Nguyen, David Chiang. 2017. Transfer Learning across Low-Resource, Related Languages for Neural Machine Translation. In *Proceedings of the Eighth International Joint Conference on Natural Language Processing, pages 296–301.*

APPENDIX

ALGORITHM 4.

```
1. Input:  directory
2. Output:  multiple small size sorted files

3. File outputfile = FilePath +  aCounter+".txt"
4. /* SAVE  file names of input directory in a local list.*/
5. File[] listOfFiles =  directory.listFiles()
6. /* from start to end of files in the list*/
7. For(i=0 to the size of the listOfFiles){
7.  aFile = listOfFiles[i].getName()
8.  If (last file) // i= size of listOfFiles
9.     end = true
10. /* 3000 is number files to be merged into one file */
11.  else if (i%3000==0)  {
12.     writeToFile (outputfile)
13.   /* create nww output file for  */
14.        aCounter =aCounter +1
15.        outputfile = path + aCounter+".txt"
16.        word_container = null
17.        word_container = new TreeSet<String>()
18. }
19. / *continue to read files  */
20. else
21. addFileContentToContainer(aFile)
22. } // end for
23. if (end) {
24. print ("Done." )
25. writeToFile (outputfile)
26. word_container = null  // free memory usage
27. }
/* helper fuction*/
1. addFileContentToContainer (aFile) {
/* variable decleration*/
2. String word = ""
3. reader = new InputStreamReader(new FileInputStream(filename),"UTF-8")
4. BufferedReader fin = new BufferedReader(reader)
4. while (Not End of File) {
5.  char s = fin.read()
6.  if s Not SpecialChar And S Not Numeric
7.            word = word + (char) s
8.    if  (s isSpace ) {
9.         word = trim(word)
10     if (word length > 1) {
11.            word_container.add (word)
12.      word = ""
13.    }
14.  }
15. } //end of while loop
16. fin.close() // close input stream
17. }
```

ALGORITHM 5.

```
1. Input:  file directory of smaller and sorted files
2. Output: sorted file
3.
4. /* variable decleation*/
5. String outputDirectory ="path"
6.  File[] listOfFiles = directory.listFiles()
7.  listForFileNames[] = new arrayList [size of listOfFiles]
8.  listForFileNames[0]="1.txt"
9.  counter =0
10. /* an empty file resides on */
11.  String file1 = listOfFiles[0]
12.  for( i=0 to size of listOfFiles)          {
13.  /*the file currently selected from the forMerge folder to for merging*/
14. String file2 = listForFileNames[i]
15. /*create next output file name which would be counter+1.txt, e.g., 2.txt */
16.  String newFiles= (counter+1) +".txt"
17. /* new file names are created and kept in a list to help processing */
18.   listForFileNames[i+1] = newFiles
19.         counter++
20.         SortedSet<String> endResultFile = new TreeSet<String>()
21.  /* at each step two files are merged throgh reading their content
22.     file1 is the one recently created, and file2 is from the list o smaller and sorted files */
23.         addFileContentToContainer (outputDirectory + file1)
24.     addFileContentToContainer (inputDirectory + file2)
25.        // write the merged file back into a new file
26.                 Writer writer = new OutputStreamWriter(new FileOutputStream(outputDirec-
tory+newFiles), "UTF-8")
27.                 BufferedWriter fout = new BufferedWriter(writer)
28.                 Iterator it = endResultFile.iterator()
29.                 while ( it.hasNext()){
30.                         fout.write((String)it.next())
31.                         fout.write(System.lineSeparator())
32.                 }
33.                 fout.close()
34.        }
```