Lab 3 – Text Processing: Apache OpenNLP

Deadline: +1 week

1 Motivation

The text must be processed before being used in an information retrieval system. Text processing usually involves several tasks, such as: tokenization, stop words elimination, part-of-speech tagging, lemmatization, stemming, or normalization. Some of these tasks are not trivial and have a strong influence on the system's behaviour.

2 Text processing

Tokenization Tokenization is a process by which the text is divided into single tokens. For example:

input: Friends, Romans, Countrymen, lend me your years;
output: [Friends, Romans, Countrymen, lend, me, your, ears]

Tokenization is not always a trivial task. Different strategies can generate different tokens, and thus different results in further text processing, such as indexing or querying. For example:

input: aren't
possible outputs: { [aren't], [are, n't], [aren, t] }

Stop words Stop words are words that do not carry any specific information about the text content. These are, e.g., conjunctions or prepositions ("a", "at", "the", "in", etc.). To generate a lists of stop-words, one may count the number of occurrences of the words in documents. It can be assumed that the words that occur frequently do not carry any important information. See an example list of English stop words.

Part-of-speech tagging (POS tagging) POS tagging is a process by which some tags are assigned to tokens. These tags inform about a role of a token (word) in a sentence (see https://www.clips.uantwerpen.be/pages/mbsp-tags). POS tagging can be useful for further information retrieval process, e.g., disambiguation:

Example 1: Cats **like** milk. → Like = verb
Example 2: The cat is white **like** milk. → Like = preposition

Lemmatization and stemming: Lemmatization and stemming are used for finding a base form of a given word.

Stemmers use heuristics for finding the base form. Thus, the method may fail to find the true base form and the outcome of the stemmer may be invalid. The most popular stemmer for English is Porter Stemmer (see https://tartarus.org/martin/PorterStemmer/def.txt). See the example outputs of the Porter Stemmer:

Example 1: Rule = **SSES→SS**	*caresses→caress*
Example 2: Rule = **IES→I**	*caresses→caress*,
	ties→ti
Example 3: Rule = **SS→SS**	*caress→caress*
Example 4: Rule = **S→**	*cats→cat*

Lemmatizers use full-morphological analysis to find a lemma of a word. In practical use, replacing a stemmer by a lemmatizer does not give much gain when it comes to English language, but lemmatizers are significantly better than stemmers for languages with a more complex morphology, e.g. Polish, German, or Spanish.

Chunking Chunking is a process by which the words are clustered into so-called chunks, according to prosodic patterns and pauses in reading, e.g.:

Input: She put the big knives on the table.
Output:

She,	Put,	the big knives,	on,	the table
NP	*VP*	*NP*	PP	*NP*

NP – noun phrase, VP – verb phrase, PP– preposition phrase

3 Apache OpenNLP

Apache OpenNLP is an open source Java library for Natural Language Processing (NLP). It supports the most common NLP tasks such as:

- tokenization,
- sentence segmentation,
- part-of-speech tagging,
- named entity extraction,

- chunking,
- parsing,
- language identification,
- coreference resolution.

OpenNLP provides pre-build models for the above tasks. These models are available for several languages, e.g., English, German, Danish, or Spanish. Additionally, OpenNLP provides a machine learning tool for training your own models. Pre-build models can be downloaded from http://opennlp.sourceforge.net/models-1.5/.

4 Programming assignment

4.1 Exercise 1

In this exercise, you are shown how to use OpenNLP for most common NLP applications.

1. Download OpenNLP.java, opennlp-tools-1.8.3.jar, and models.zip from the lab directory. Create a Java project using any IDE. The **OpenNLP.java** file is a template for the following tasks. It contains several methods for text processing (e.g., **private void tokenization()**) which must be finished. Unpack the **models.zip** archive to the root directory of your project. This archive contains several models that have to be loaded by OpenNLP in order to perform the processing. Consider the **run** method. It invokes subsequent methods (**languageDetection**, **sentenceDetection**, etc.). If you wish to run and verify results of a single step only, just comment out some of the calls.

2. **Language detection**: OpenNLP provides a tool for language detection. See here for the list of supported languages.

   (a) As you may notice, a **LanguageDetectorModel** object is created. It uses a **langdetec-183.bin** model for language detection.

   (b) Create a **LanguageDetectorME** object.

   (c) Call a **predictLanguage** (or **predictLanguages**) method to perform language detection for a given text. This method returns a **Language** object. It contains information about in which language(s) the text is (are) written in along with the confidence value(s).

   (d) Run this method for the example texts provided in the code. Observe the relation between a predicted language(s) (and confidence values) and length of an input. How does a probability change if an input text is written in two languages?

3. **Tokenization**:

   (a) Use instances of a **TokenizerME** a **TokenizerModel** classes. When creating a **TokenizerModel** object, load and pass an **en-token.bin** model.

   (b) Use a **tokenize** method to tokenize input strings provided in the code. In addition, use a **getTokenProbabilities** method to obtain probability values.

   (c) Compare the obtained tokens for each text. Are the outcomes different? Why?

   (d) Load a model for German language: **de-token.bin** (but you can try any other model from http://opennlp.sourceforge.net/models-1.5/).

   (e) Run the method and compare the obtained results with the outcomes for the **en-token.bin** model. Are there any differences?

4. **Sentence detection**:

   (a) Use a **SentenceModel** and a **SentenceDetectorME** objects with a **en-sent.bin** model to detect sentences in a given phrase.

   (b) Find a proper method of the **SentenceDetectorME** object for determining sentences in a provided text and the corresponding probabilities.

   (c) Call these methods for the example texts provided in thwe code and print the results.

   (d) Do you see any invalid sentence segmentation outcome?

   (e) Observe what happens with short sentences like "Hi.".

   (f) Add some punctuation or question marks (or double them, e.g., "??"). Run the method. How did these modifications affect the results?

5. **Part-of-speech tagging**: Part-of-speech (POS) tagging allows detecting POS for each token in a given sentence. The list of available POS tags, their meaning, and examples, is available here.

   (a) Use a **PPOSModel** and a **PPOSTaggerME** objects with an **en-pos-maxent.bin** for POS tagging.

   (b) Determine part-of-speech for sentences "Cats like milk" and "Cat is white like milk" provided in the code.

   (c) Observe the obtained tags. Are these tags correct? Consider the word "like". Is the tag assigned correctly in both cases?

6. **Lemmatization and stemming**:

   (a) Use a **PorterStemmer** and a **DictionnaryLemmatizer** objects and an **en-lemmatizer.dict** model to find the base form for the tokens given in the exercise.

(b) Compare the results of the the stemmer and the lemmatizer. Do you see some differences? What happened with the word "are"? Why the lemmatizer requires POS tags?

(c) Check what happens when a given token does not exists in the dictionary (provide a random string). Compare the results of the stemmer and lemmatizer.

7. **Chunking**:

   (a) Use a **ChunkerModel** and a **ChunkerME** objects and a **en-chunker.bin** model to chunk sentences into clusters. Why are the POS tags required?

   (b) Run the method and observe the results. How are the chunks marked (see https://www.clips.uantwerpen.be/pages/mbsp-tags)? What does the "B-" preffix mean? What does the 'I-" preffix mean? How many chunks do you see? Do you think the obtained results are correct?

8. **Named entity extraction**: The Name Finder can be used to detect named entities and numbers in texts. There are different models for different entities. OpenNLP provides pre-build models for, e.g., finding dates, locations, names of people, or names of organizations. We will focus on finding names of people.

   (a) Use a **TokenNameFinderModel** and a **NameFinderME** objects. To identify names of people, use an **en-ner-person.bin** model.

   (b) Find people in the provided paragraph. Are the results correct?

   (c) Use an **en-ner-xyz.bin** model and run the method again. What kind of entities do you think the model seeks for?

5 Exercise 2

Given is a collection of 20 movie descriptions (movies.zip). These are text files. Your task is to process these files, using OpenNLP, and compute/derive the following statistics/features:

For each movie:

- number of sentences,
- number of tokens,
- number of **unique** stems (stemming),
- number of **unique** words (lemmatization),
- list of people, locations, organizations.

Overall statistics:

- percentage number of adverbs,
- percentage number of adjective,
- percentage number of verbs,
- percentage number of nouns.

1. Download a MovieReviewStatistics.java template, which reads data from files, prints, and stores statistics. Your task is to complete two methods (see **TODOs**): **initModelsStemmerLemmatizer**, and **processFile**. The first method should load required models (for English language!) and create Stemmer and Lemmatizer objects. Use the Porter Stemmer. The **processFile** method should compute/update the statistics. Before you complete these methods, run the program. You should see that a *statistics.txt* file was generated. For each movie, it should include entries in the following form:

   Movie: Some movie

   Sentences: 10

   Tokens: 100

   Stemmed forms (unique): 58

   Words from dictionary (unique): 30

   People: John, Mr. Smith

   Locations: Poland

   Organizations: XYZ

   The overall statistics are included at the end of the file. Obviously, the code is not finished, thus there are no results. All these logs are also printed to the console when executing the program. Now, complete the code. Pay attention to **TODOs** and commented hints.

2. Run the code. Compare the computed statistics (the sum does not equal 100%; why?) with these for English conversation language from "Longman Grammar of Spoken and Written English":

   Adverbs 0.5%

   Adjectives 2.5%

   Verbs 12.5%

   Nouns 15%

   Think about why there is such a difference in the adjectives percentage between the obtained results and the results for general English language.

5.1 Useful (and sufficient) imports

OpenNLP imports:

```
import opennlp.tools.chunker.ChunkerME;
import opennlp.tools.chunker.ChunkerModel;
import opennlp.tools.langdetect.Language;
import opennlp.tools.langdetect.LanguageDetectorME;
import opennlp.tools.langdetect.LanguageDetectorModel;
import opennlp.tools.lemmatizer.DictionaryLemmatizer;
import opennlp.tools.namefind.NameFinderME;
import opennlp.tools.namefind.TokenNameFinderModel;
import opennlp.tools.postag.POSModel;
import opennlp.tools.postag.POSTaggerME;
import opennlp.tools.sentdetect.SentenceDetectorME;
import opennlp.tools.sentdetect.SentenceModel;
import opennlp.tools.stemmer.PorterStemmer;
import opennlp.tools.tokenize.TokenizerME;
import opennlp.tools.tokenize.TokenizerModel;
import opennlp.tools.util.Span;

import java.io.File;
import java.io.IOException;
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
```

MovieReviewStatistics imports:

```
import opennlp.tools.lemmatizer.DictionaryLemmatizer;
import opennlp.tools.namefind.NameFinderME;
import opennlp.tools.namefind.TokenNameFinderModel;
import opennlp.tools.postag.POSModel;
import opennlp.tools.postag.POSTaggerME;
import opennlp.tools.sentdetect.SentenceDetectorME;
import opennlp.tools.sentdetect.SentenceModel;
import opennlp.tools.stemmer.PorterStemmer;
import opennlp.tools.tokenize.TokenizerME;
import opennlp.tools.tokenize.TokenizerModel;
import opennlp.tools.util.Span;

import java.io.File;
import java.io.IOException;
import java.io.PrintStream;
import java.nio.file.Files;
import java.text.DecimalFormat;
import java.util.Arrays;
import java.util.Comparator;
import java.util.HashSet;
import java.util.Set;
import java.util.logging.Level;
import java.util.logging.Logger;
```