Count

Brian Clark

November 22, 2019

An NLP library and CL app for counting words, sentences, and paragraphs.

Contents

1 Summary

`count` is a Java library that allows analyzing English text to determine the number of words, sentences, and paragraphs in an input sample.

It also provides a command line utility to run this analysis on a text file (see Usage). `count` uses Maven to organize dependencies, build, testing, and packaging tasks (see Build).

2 Usage

2.1 CLI

```
java -jar <count-jar> -file <source-file>
```

`<count-jar>`: Enter the path to the `count` JAR. If building using Maven, this will be located in the `target/` subdirectory.

`<source-file>`: Enter the path to a plain text file you want to run the `count` analysis on.

2.2 Testing

Tests can be run using `mvn verify`. This will run both the unit and integration test suites.

3 Build

`count` can be run either from a distributed tarball or by building from source.

3.1 Source

1. Clone the repository from `git@github.com:bclarkx2/count`
2. Build the project using `mvn install`
3. Run the project using the JAR file generated at `target/count-<version>.jar`

3.2 Tarball

1. Untar the `count-<version>.tar.gz` to a file location (e.g. by running `tar -xzvf count-<version>.tar.gz`
2. Run the project using the JAR file generated at `count/target/count-<version>.jar`

4 General Approach

I approached this task with two objectives in mind:

1. Define the interfaces of the necessary components, and then iterate on each to improve. Once the boundaries are defined, it will be easier to iteratively attack the well-defined problem and try to come up with better solutions to these tricky sorts of problems.

2. Use the principle of composition. Each component should be easily re-used as a part of another component while building the processing pipeline.

5 Project Structure

5.1 Directory Structure

`/src` - all of the source material for the project
`/src/main/java` - java source code
`/src/main/resources` - resources used during execution
`/src/main/assembly` - configuration used to generate a compressed project assembly
`/src/test` - root for all test code and resources at the unit level
`/src/it` - root for all test code and resources at the integration level
`/site/tex` - source material for this and other documentation `/target` - output build directory

5.2 Code organization

Here is a description of the basic codeflow of the command line program.

1. All the Java class files and dependencies are packaged into the `count` JAR file, with the `Runner` class set as the main class (from the `info.clarknet.count.runner` package.

2. When the main method of the `Runner` class is invoked, the command line flags are parsed to determine which source file and which type of Counter should be used.

3. Then a `Sample` class is constructed based on the source file path provided via command line. This `Sample` class abstracts away the process of reading in text from an IO source, and could potentially in the future store metadata about the text contained within it.

4. The `Sample` object is fed as an input to an instance of a class implementing the `Counter` interface (e.g., the counter!). The counter's job is to turn this `Sample` input into a `CountResult` object that holds data about the count of words, sentences, and paragraphs within the `Sample`. The `Counter` object will make use of any of the classes defined in the

`info.clarknet.counter.count` package to accomplish this goal, mixing and matching them as necessary to get the best result.

5. Finally, the `CountResult` object will be passed to a `CountReporter` object. This `CountReporter` will be instantiated with all the information it needs to deliver a full report on the `CountResult` to the output destination of choice. (For example, the `StreamReporter` implementation will write a report to the standard output stream.

6 NLP Heuristics

6.1 Available Counters

The `BasicCounter` class is the current best attempt at implementing a counter. I developed all the heuristics used in this class. `OpenNLPCounter` is provided as a sort of reference implementation using parts of the Apache OpenNLP library.

6.2 Word detection

The `info.clarknet.counter.count.token` package contains the code designed to identify words in the given text. The `Tokenizer` interface describes the behavior that any Tokenizer is responsible for, namely, transforming one string or stream of strings into a "tokenized" stream of strings.

In order to count words, I defined a `WhitespaceTokenizer` class that implements `Tokenizer`. This class takes advantage of the fact that English words are typically separated by one or more whitespace characters. The algorithm used is as follows:

1. Trim the input string of leading and trailing whitespace.
2. If there are no remaining non-whitespace characters, return an empty stream.
3. Split the string on a greedy whitespace regular expression (this will capture as many whitespace characters as are found between non-whitespace characters)
4. Return a stream of the results.

Counting the elements in this resulting stream yields a pretty good approximation of the number of words in the input text.

6.3 Sentence detection

Similarly to word detection, the sentence detection code is largely contained within the `info.clarknet.counter.count.sentence` package, with the `SentenceDetector` interface defining the public behavior.

The `NaiveSentenceDetector` implements my first pass at counting sentences. It has two parts to the logic it uses.

6.3.1 PunctuationTokenizer

The first pass at sentence detection involves running a PunctuationTokenizer over the input text. This involves first using a `WhitespaceTokenizer` to split the string into whitespace-separated tokens. Then, we map each whitespace token into one or more subtokens using several rules. Quote characters at the beginning and end of the token are separated as their own tokens, sentence-ending punctuation characters such as ".", "?", and "!" are removed from the end of the token, and any number of hyphens at the end of the token will be treated as a single token and removed from the end. Tokens that fully match a known abbreviation will be treated as a single token and not broken down any further.

Hyphenated words are treated as a single token. For purposes other than simply counting words, it can be useful to confitionally split hyphenated words into multiple tokens. However, for our purposes here it is easier to keep them as one.

The end result of this process is a stream of tokens that are more useful to determining where the sentence boundaries lie.

6.3.2 `NaiveSentenceDetector`

The `NaiveSentenceDetector` simply counts the number of sentence-ending tokens within the stream of punctuation-separated tokens generated from the `PunctuationTokenizer`.

This works fairly well at identifying the number of sentences. The largest shortcoming is in handling setences that contains quoted sections within the sentence where one of these punctuation tokens appear inside the quoted section.

6.3.3 `FragmentSentenceDetector`

My plan to tackle this challenge of nested, quoted sentence parts is to create the `FragmentSentenceDetector` implementation. Although this is not fully finished, the basic idea is to parse a stream of punctuation tokens into a stream of sentence fragments. As we are parsing the stream, a new fragment will begin whenever the previous fragment reaches a sentence ending punctuation token OR when a begin quote token is encountered. If a fragment is currently open when a new fragment is encountered, the second fragment will be added as a subfragment to the first. After parsing the whole stream, we can look at the structure of the various fragments to determine a sentence count while safely ignoring fragments that don't count as sentences because they were in nested quotes.

6.4 Paragraph detection

The `info.clarknet.counter.count.paragraph` package contains all the classes necessary for counting the number of paragraphs in a text sample.

The `LinebreakParagraphDetector` implements this behavior in a fairly straightforward manner. The input string is first trimmed of leading and trailing whitespace, and then split on the System-defined line separator. We then iterate over the resulting string array, adding each line to a StringBuilder as we go. As soon as we encounter a blank line, we concatenate the current contents of the StringBuilder and add the result to our list of identified paragraphs. We then wipe out the StringBuilder and continue iterating, beginning to build up the StringBuilder once again as soon as we read the next non-blank line. Once we finish processing each line, we dump any remaining contents of the StringBuilder as the final paragraph in our list of paragraphs.

Counting the size of this resulting list gives us the estimate for the number of paragraphs in the input string.

7 Areas of future growth

The following are areas to be developed in the future:

- FragmentSentenceDetector: This is the next improvement I have planned (See the `FragmentSentenceDetector` section).
- Abbreviation detector: I would like to improve this beyond just checking against a set of known abbreviations.
- Support for single quotes.
- Large file size robustness: The library currently depends on the entire sample text being able to fit into memory as a single `String` object. I would like to implement a line-based approach that allows analyzing extremely large input files as well.
- i18n and string encodings.
- Investigate parallel processing (e.g. counting sentence in different paragraphs in parallel tasks).