Raphael Braun
Ayesha Feroz
Kyowon Jeong
Matteo Pilz
Lukas Ruppert
Arslan Siraj
Faezeh S. Zakeri

Eberhard Karls
Universität
Tübingen

Programming in C++

Sheet 5

Submission date: 12.10.2022

5.1 Word Histograms (25P [20 + 1 + 2 + 2] + 30P + 30P + 15P)

C++

A simple way to analyse a long text is to find the unique words and count how often they appear in the text. Consider the following example:

```
IS THE SUN IN THE SKY OR IS THE SUN IN SPACE
```

We get a histogram by going through all words. If the word is new we assign it the count 1, if we have seen it before we increase its count by 1. This way we will end up with all unique words and how often they occur. This is shown in Table 1a.

Table 1: Histogram Example

Word	IN	IS	OR	SKY	SPACE	SUN	THE
Count	2	2	1	1	1	2	3

(a) Not normalized

Word	IN	IS	OR	SKY	SPACE	SUN	THE
Probability	$\frac{2}{12}$	$\frac{2}{12}$	$\frac{1}{12}$	$\frac{1}{12}$	$\frac{1}{12}$	$\frac{2}{12}$	$\frac{3}{12}$

(b) Normalized

The longer the texts are the higher the count numbers will be. To compare long texts it therefore helps to normalize the histogram by dividing the counts with the total number of words in the text. With that you get the probability for randomly picking the unique word. This is shown in Table 1b.
In this exercise you will implement a normalized word histogram class using `std::map`, which will allow you to analyze and compare long texts.

The `main.cpp` file already loads the words of 4 books into lists. You can uncomment the code that relies on 5.1.b, 5.1.c and 5.1.d in order to run your implementations on those books.
Book0 and book1 are law books, book2 and book3 are books about programming languages.

a) Implement all methods in `submission/histogram.cpp` that are marked with `// TODO 5.1.a)`. Check the comments in `submission/histogram.h` to see what the methods are supposed to do.

Hint: Take a look at the docmumentation of `std::map` to see how `std::map` can help you to solve those tasks.

b) We would like to extract the `n` most common words of the text from the histogram in descending order. You can do this by constructing a `std::multimap` which uses the probabilities as key and the words as values. The `std::multimap` is sorted, so you just have to extract the last `n` entries to get the most common words.

Add a method to the Histogram class with the following signature:

```
std::vector<std::pair<double, std::string>> most_common_words(
                                                    unsigned int n_words) const;
```

Hint:

- Google might help you to figure out how to reversely iterate through a map in C++.
- The values stored in a `std::multimap` are represented as `std::pair<KeyType, ValueType>`. Dereferencing the iterator should directly gives you a `std::pair<double, std::string>`.

c) We would like to be able to compare the Histograms of two different texts. Add another method with signature:

```
double dissimilarity(const Histogram &other) const;
```

This method should compute the dissimilarity (difference) of the histogram to the `other` histogram by summing up the following summands:

- 1 minus the ratio between the number of shared words and all the words in this histogram (A).
- 1 minus the ratio between the number of shared words and all the words in the other histogram (B).
- For all words that appear in both histograms dissimilarity is defined as the absolute difference between the words probability in both histograms. The dissimilarity of all those words is added.

The dissimilarity D therefore is:

A : This histogram

B : The other histogram

$S = \{w | w \in A \wedge w \in B\}$: Shared words

$$D = \left(1 - \frac{|S|}{|A|}\right) + \left(1 - \frac{|S|}{|B|}\right) + \sum_{w \in S} |A.\texttt{probability}(w) - B.\texttt{probability}(w)|$$

d) Add a method that accepts a list of histograms as input and returns the index of the histogram with the lowest dissimilarity to the histogram from which the method is called. Its signature has to be:

```
size_t closest(const std::vector<Histogram> &candidates) const;
```