Lotte and Annette: A Framework for Finding and Exploring Key Passages in Literary Works

Frederik Arnold and **Robert Jäschke**
Humboldt-Universität zu Berlin
{frederik.arnold, robert.jaeschke}@hu-berlin.de

Abstract

We present an approach that leverages expert knowledge contained in scholarly works to automatically identify key passages in literary works. Specifically, we extend a text reuse detection method for finding quotations, such that our system *Lotte* can deal with common properties of quotations, for example, ellipses or inaccurate quotations. An evaluation shows that Lotte outperforms four existing approaches. To generate key passages, we combine overlapping quotations from multiple scholarly texts. An interactive website, called *Annette*, for visualizing and exploring key passages makes the results accessible and explorable.

1 Introduction

Identification of key passages in nonfiction has long been a topic of research. For example, in the context of text summarization to identify sentences and passages which contain key arguments (Paice, 1980). While there has been a lot of progress for nonfiction (Yao et al., 2017), there are no working solutions for fiction.

In this paper, we present a first step towards a system to automatically identify key passages in fiction. We understand *key passages* as passages that are particularly important to expert readers, following a general definition of "key words" (Scott and Tribble, 2006). We leverage the expert knowledge contained in scholarly works to automatically identify potential key passages. Authors of scholarly works use different types of citations to refer to original works, for example, quotations or paraphrases. We adapt existing methods for text re-use detection (Grune and Huntjens, 1989) such that our system can deal with common properties of quotations such as ellipses or unclean quotations, for instance, missing words or spelling mistakes. On top of that, our system works independently of the order of quotations and can handle multiple quotations of the same text. To generate key passages, we combine overlapping quotations from multiple scholarly works.

Our contributions are *Lotte*, an algorithm and Python tool for quotation detection in fictional texts and *Annette*, an interactive website for visualizing and exploring key passages. This makes key passages available at a larger scale and in structured form and opens up many new opportunities for analyses in literary studies and the praxeology of literary studies.

This paper is organized as follows: In Section 2, we provide an overview on related work. In Section 3, we present our approach to finding key passages. In Section 4, we describe our evaluation setup including four existing systems and in Section 5 we show how our approach outperforms them. Finally, in Section 6, we present our tool for visualizing and exploring key passages.

2 Related Work

Quotation detection can be regarded as a kind of text reuse detection, which is frequently applied for plagiarism detection (Hoad and Zobel, 2003). There, the goal is to find quotations and citations without proper attribution. In our case, we assume proper attribution and focus on the step of *finding* and *linking* quotations.

Several tools for different use cases try to solve similar or related problems. For example, *BLAST* aligns biological sequences (Altschul et al., 1990) and has been adopted for text reuse detection (Vesanto et al., 2017a,b). Copyfind (Bloomfield) is an open source tool for comparing documents written in C++. While *Passim* (Smith et al., 2014) and *TextMatcher* (Reeve, 2020) are simple text reuse detection tools, *TRACER* (Büchler, 2016) is an elaborate framework consisting of around 700

Proceedings of the Workshop on Natural Language Processing for Digital Humanities (NLP4DH), pages 55–63
co-located with ICON-2021, December 16-19, 2021, Silchar, India.
https://doi.org/10.26615/978-952-94-5833-2_007

algorithms. *SIM* (Grune and Huntjens, 1989) finds lexical similarities in source code and natural language texts, originally built to find duplicate code in large code bases. The original idea worked well enough to be used to find copied work in student submissions. SIM works with a number of programming languages and can easily be extended to work with new languages by providing a lexical description. *Sim_text* is a version of SIM for checking duplicates in natural language texts. Based on SIM, *similarity texter (SimT)* is a tool for text comparison written in JavaScript by Kalaidopoulou (2016).

For various reasons, these tools are not appropriate for detecting key passages. For example, TextMatcher only finds quotations that appear in the same order in both texts. None of the tools can find multiple quotations of the same text. In Section 5 we evaluate BLAST, Copyfind, SimT, and TextMatcher and compare them against our approach. We did not evaluate TRACER, as we could not manage to extract exact matches. Passim is the only system we could not get to work at all as its dependencies were no longer available.

A website for visualizing (literal) citations of Shakespeare's works has been presented by Miller. It visualizes how often each line from every play has been cited in JSTOR's journal collection. The website is limited to the visualization of the citation frequency of each line and does not offer any functionality to explore the source of citations.

3 Lotte – A Text Reuse Detection Tool

In this section, we describe our approach for identifying quotations which solves the following task:[1] Given a *source* and a *target* text, it finds all instances where the target text contains some part of the source text.

Our approach is based on a modified and extended version of Sim_text by Grune and Huntjens (1989). The original implementation is written in C while our reimplementation is in Python. Reimplementing the algorithm allowed us to integrate extensions for properly handling specific properties of quotations which are not covered by Sim_text.

The algorithm works in five main steps which we describe in the sequel. Table 1 shows two simplified example texts which consist of words only without any punctuation except for periods and one ellipsis in the target text. The numbers on the left are the positions of the words (w_1 in the source text is at position 0, w_2 at position 1, w_8 at position 7, etc.).

Source text
^{0} w_1 w_2 w_3 w_4 w_5 w_6 w_7 w_8. w_4
^{9} w_5 w_6 w_7 w_8 w_9. w_{10} w_{11} w_{12}
$^{17}w_{13}$ w_{14} w_{15} w_{16} w_{17} w_{18} w_{19}
$^{24}w_{20}$ w_{21} w_{22} w_{23} w_{24} w_{25} w_{26}
Target text
$^{50}w_1$ w_2 w_3 w_{11} w_{12} w_{13} [...] w_{23} w_{24}
$^{59}w_{25}$ w_{26}. w_{30} w_{31} w_{32} w_{33} w_{34} w_4
$^{67}w_5$ w_6 w_7 w_8 w_9. w_{10} w_{11} w_4 w_5 w_6

Table 1: Example source and target texts.

Word sequences	Starting positions
w_1 w_2 w_3	[0]
w_2 w_3 w_4	[1]
w_4 w_5 w_6	[3, 8]

Table 2: Some word sequences with starting positions of the source text.

3.1 Step 1: Tokenize Text

Both texts are cleaned and tokenized and sequences like '...', '[...]' and other possible variants of ellipses-indicating characters are masked so they can later be identified easily. Punctuation indicating the end of a sentence is also masked for the same reason. All other special characters and numbers are removed. Finally, the text is tokenized using white space characters. The main improvement to Sim_text is the masking of characters which carry information needed later.

3.2 Step 2: Initial Positions

A mapping of word sequences to starting positions for the source text is created. The initial sequence length is currently hard-coded to three as it worked best in our tests. The value can easily be changed but a smaller value results in too many initial matches which will be removed later anyway because of the minimal length cutoff for results. Table 2 shows examples for word sequences and their starting positions. The sequence w_1 w_2 w_3 starts at position 0, sequence w_2 w_3 w_4 at position 1, etc. The same sequence can appear multiple times and

[1] The source code is licensed under the Apache License 2.0 and available at https://scm.cms.hu-berlin.de/schluesselstellen/lotte.

Source text	Target text
0	[0]
3	[67]
8	[67]

Table 3: Some source text starting positions with corresponding target text starting positions.

therefore might have multiple starting positions. This handling of sequences that appear multiple times is the main improvement to Sim_text.

3.3 Step 3: Forward References

A table of forward references, that is, a mapping of starting positions in the source text to a list of starting positions in the target text is created. For example, the sequence $w_1\ w_2\ w_3$ which starts at position 0 in the source text can also be found in the target text starting at position 0.

As Sim_text only considers exact matches, we improved this to use MinHash Locality Sensitive Hashing (Slaney and Casey, 2008) and Levenshtein distance (Levenshtein, 1966) to find the best matching sequence. Our algorithm first gets a list of all possible matches above a similarity threshold of 0.95. From that list, it then selects the best match with a normalized Levenshtein distance[2] equal or greater 0.9. These thresholds were optimized using expert knowledge (cf. Section 4.2).

3.4 Step 4: Extend Initial Matches

The initially three token long matches are extended forwards and backwards to match longer sequences, if possible. For example, in Table 1 the initially matched sequence $w_4\ w_5\ w_6$ can be extended forward to match $w_4\ w_5\ w_6\ w_7\ w_8\ w_9\ w_{10}\ w_{11}$. Backwards extension is needed to handle certain edge cases occurring due to ellipses or mismatches of tokens. Sim_text does neither include backwards extension nor handling of ellipses or mismatches of tokens. Our algorithm also uses the normalized Levenshtein distance for token matching.

3.5 Step 5: Reprocess Found Matches

Step 4 extends the initial matches in a relatively conservative manner. A more aggressive approach would lead to too many false positives. This means that the quality of the matches can be further increased by reprocessing the intermediate matches. This is implemented in the following novel steps.

Start	End	Match segments
50	61	$w_{11}\ w_{12}\ w_{13}$
12	23	$w_{11}\ w_{12}\ w_{13}$
98	113	$w_{23}\ w_{24}\ w_{25}\ w_{26}.$
28	44	$w_{23}\ w_{24}\ w_{25}\ w_{26}.$
9	25	$w_4\ w_5\ w_6\ w_7\ w_8.$
65	79	$w_4\ w_5\ w_6\ w_7\ w_8$
26	53	$w_4\ w_5\ w_6\ w_7\ w_8\ w_9.\ w_{10}\ w_{11}$
65	91	$w_4\ w_5\ w_6\ w_7\ w_8\ w_9.\ w_{10}\ w_{11}$

Table 4: Intermediate results after Step 4. First line of each pair (red) is the match segment from source text and the second line (blue) is the match segment from the target text.

Table 4 shows some of the intermediate results after executing Steps 1 to 4. The first line of a pair is the match segment from the source text and the second line is the match segment from the target text. The two numbers at the beginning of a line correspond to the first and last character's position of a match in the original text, respectively.

Neighbouring Matches The first improvement is to merge neighbouring matches. The intermediate matches are sorted in order of appearance in the source text. They are then checked for matches that appear neighbouring in the target text and in the source text and are not further apart than a certain number of tokens. If there is an ellipsis between the two matches in the target text, the number of tokens between the matches can be greater.

Overlapping Segments We remove matches with overlapping target match segments. In Table 4, the last two matches completely overlap in the target text. This means that one of the matches has to be removed. In such a case only the longer one will be kept.

Short Matches The remaining matches are checked for matches which are shorter than a certain length, which can be defined by the user. In our case, we only keep matches which are five words or longer. All other matches are removed.

Sentence Boundaries Finally, we check for matches that cross sentence boundaries. This happens in a number of cases where after a match the

[2] https://github.com/maxbachmann/RapidFuzz

Start	End	Match segments
50	113	w_{11} w_{12} w_{13} w_{14} w_{15} w_{16} w_{17} w_{18} w_{19} w_{20} w_{21} w_{22} w_{23} w_{24} w_{25} w_{26}
12	44	w_{11} w_{12} w_{13} [...] w_{23} w_{24} w_{25} w_{26}.
26	44	w_4 w_5 w_6 w_7 w_8 w_9.
65	83	w_4 w_5 w_6 w_7 w_8 w_9.

Table 5: Final results. For both matches, the first part (red) is from the source text and the second part (blue) is the match segment from the target text.

source and target text continue with the same words by chance. We check for matches which end with a sentence delimiter (., ;, ! and ?) followed by one or two words. In such cases the words after the delimiter are removed. The final results are shown in Table 5.

4 Experiments

4.1 Datasets

We evaluate our approach on two literary works, *Die Judenbuche* by Annette von Droste-Hülshoff (1979) and *Michael Kohlhaas* by Heinrich von Kleist (1978), with 44 and 49 interpretive scholarly articles, respectively.[3] The texts were annotated in the ArguLIT project (Winko, 2017–2020) using TEI/XML (TEI Consortium, eds.). The corpus contains annotations for quotations of different types, for example, quotations from the primary literary work, other literary works, or other scholarly works. Only clearly marked quotations, that is, with quotation marks, were annotated. For the purposes of this evaluation, we are only interested in *quotations from the primary literary work*. Table 6 shows the number of articles and quotations from the primary literary work with a length of five or more words ("gold items").

We limit the experiments to finding matches of five or more words because none of the approaches works for very short matches. It would be possible to find shorter matches but it introduces too much noise. The limit is based on the distribution of all word *n*-grams which have a frequency of at least two (cf. Table 7). The counts are calculated after removing special characters and only the longest sequence is counted, for example, for a 7-gram, the 3- and 4-sub-grams are not counted again. *Die Judenbuche* contains two 5-grams, one 6-gram, and one 7-gram which appear twice. This is few enough to not introduce too much noise. In the case of *Michael Kohlhaas*, which is twice as long as *Die Judenbuche*, the *n*-gram counts do not as clearly support a limit of five or more words. We decided to keep the limit but this could be improved in the future. For example, as a first step, Lotte could report ambiguous cases. In the longer term, we will develop methods for extracting quotations shorter than five words and handling ambiguous cases.

Literary work	Die Judenbuche	Michael Kohlhaas
Scholarly articles	44	49
Gold items (≥ 5 words)	1 235	1 349
Quotations with ellipses	206	262
Literary text characters	102 477	221 097
Scholarly articles characters	2 650 095	2 778 528

Table 6: Basic statistics for *Die Judenbuche* and *Michael Kohlhaas*.

n-gram	Frequency 2		3		4		5		> 5	
	J	K	J	K	J	K	J	K	J	K
3	130	752	14	114	3	27	1	4	2	7
4	21	176	1	21	0	3	0	2	0	0
5	2	55	0	7	0	1	0	0	0	0
6	1	11	0	1	0	0	0	0	0	1
7	1	3	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0
9	0	1	0	0	0	0	0	0	0	0
10	0	2	0	0	0	0	0	0	0	0

Table 7: The *n*-gram counts for *Die Judenbuche* and *Michael Kohlhaas*.

4.2 Setup

For each approach, we try to select parameters as close as possible to those of our approach. Minimal match length is always set to 5. Lotte's thresholds and parameters were optimized on the corpus for *Die Judenbuche*. The results will show that the approach performs equally well on unseen texts.

BLAST There are several parameters but none really correspond to those of the other approaches.

[3] For the sake of brevity, we will reference *Die Judenbuche* and *Michael Kohlhaas* with J and K.

	BLAST	Copyfind	Lotte	SimT	TextMatcher
Order independent	✓	✓	✓	✓	-
one-to-many matching	-	-	✓	-	-
Fuzzy matching	✓	-	✓	-	-
Skip words	-	✓	✓	-	✓
Ellipsis handling	-	-	✓	-	-

Table 8: System functionality comparison.

So we use the defaults and remove short matches in a post-processing step. BLAST requires a mapping from characters to DNA sequence blocks. Using the provided mapping for English with space worked better than using a mapping based on the most frequent characters in German.

Copyfind We ignore letter case, numbers and punctuation. We allow up to two non-matching words between perfectly matching phrases and a minimum of 80 % matching words for a phrase to be considered a match.

Lotte We use the following parameters: A look-back limit of 10, a look-ahead limit of 3, a maximum merge distance of 2, and a maximum merge distance for ellipses of 10. We ignore letter case, numbers, punctuation, and replace umlauts.

SimT We ignore letter case, numbers and punctuation and replace umlauts.

TextMatcher Again, there are several parameters but none really correspond to those of the other approaches. We set threshold and cutoff to 0 and leave the default value of 3 for n-gram size. We also remove short matches in a post-processing step.

Table 8 shows a comparison of the functionality of each approach. TextMatcher is the only system that does not support order-independent matching, that is, only matches appearing in the same order in both texts will be found. One-to-many matching, that is, matching a sequence in the source text with multiple sequences in the target text is only supported by Lotte. Fuzzy matching is supported by Lotte and BLAST. Copyfind, Lotte, and TextMatcher can skip words, that is, a sequence can still be a match even if there is a mismatch between individual words. Lotte is the only system that explicitly handles ellipses. Processing 44 scholarly works for *Die Judenbuche* with Lotte takes around five minutes on an Intel Core i9-9880H CPU.

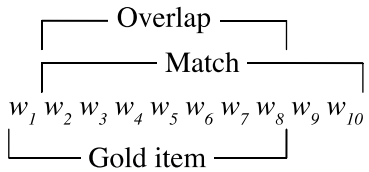

Figure 1: Calculation of precision (|Overlap| / |Match|), recall (|Overlap| / |Gold item|), and F_1-score based on the overlap between a match and a gold item.

4.3 Evaluation

For the evaluation, we asses the performance of all approaches by averaging precision, recall, and F_1-score of each match and gold item. Figure 1 illustrates the calculation. Internally, we use character counts for the calculation. This ensures that the results of all approaches are comparable and is necessary in case an approach does not respect token boundaries and returns incomplete words. Matches which cover multiple gold items are punished by taking the average precision. Analogously, gold items which are partly covered by multiple matches are punished by taking the average recall.

5 Results

5.1 Performance Comparison

Table 9 shows the performance of the approaches in the top section. The bottom section shows different variants of Lotte which we discuss in Section 5.3.

For *Die Judenbuche*, Lotte outperforms the other approaches with an F_1-score of 0.86. Copyfind performs second best (0.79), closely followed by SimT (0.76). SimT's precision is highest with 0.91.

For *Michael Kohlhaas* the results look different. Lotte achieves the highest recall of 0.90, but SimT performs best with the highest precision of 0.83 and an F_1-score of 0.79.

5.2 Error Analysis

To better understand the differences in precision between the approaches and the lower precision

Approach	Die Judenbuche			Michael Kohlhaas		
	Precision	Recall	F_1	Precision	Recall	F_1
BLAST	0.59	0.61	0.60	0.37	0.59	0.45
Copyfind	0.85	0.75	0.79	0.76	0.79	0.78
SimT	**0.91**	0.64	0.76	**0.83**	0.74	**0.79**
TextMatcher	0.69	0.37	0.48	0.68	0.42	0.52
Lotte	0.82	**0.90**	**0.86**	0.70	**0.90**	0.78
Lotte-Base	0.96	0.29	0.45	0.84	0.26	0.40
+ OI	0.91	0.64	0.75	0.84	0.74	0.79
+ OI+otm	0.90	0.72	0.80	0.83	0.79	0.81
+ Fuzzy	0.88	0.83	0.85	0.79	0.84	0.81
+ Skip	0.85	0.84	0.84	0.75	0.85	0.79
+ Ellipsis	0.90	0.74	0.82	0.83	0.82	0.82

Table 9: Precision, recall, and F_1-score for *Die Judenbuche* and *Michael Kohlhaas*.

of Lotte, we analyze the different types of false positives as shown in Table 10. The second column shows the total number of false positives, followed by the counts for three relevant types of false positives. For example, out of the 279 false positive matches found by Lotte for *Die Judenbuche*, 64 are type *other*, that is, there is a match in our gold annotations which was not annotated as a quote from the primary literary work but some other text, for example, other literary works or scholarly works. For example, *Die Judenbuche* quotes the Bible and that same quote is quoted in a scholarly work and attributed to the Bible by our annotations but, of course, Lotte counts it as a match. 31 are of type *short*, that is, a match with five words or more was found but the corresponding gold item is only four words long. *O+S* is the combination of the two previous cases. The remaining false positives do not belong to any category.

Comparing the numbers for Copyfind, Lotte and SimT, we find that for both literary works, SimT and Copyfind have less false positives of the three types. Counting these as true positives Lotte's precision would improve relative to the other two approaches.

Another reason for the high number of false positives is that a large number of quotations are not annotated at all because they are not correctly highlighted (e.g., with quotation marks). This issue is worse for Lotte because of the improved handling of quotation-specific properties which leads to a higher number of false positives which are actually true positives but are missing in our data. The false positives which do not belong to any of the

Approach	Total		Other		Short		O+S	
	J	K	J	K	J	K	J	K
BLAST	227	646	30	37	33	45	0	1
Copyfind	174	232	46	22	24	29	0	0
Lotte	279	404	64	47	31	47	2	0
SimT	128	186	40	22	12	25	0	0
TextMatcher	130	112	14	1	4	13	0	0

Table 10: False positives counts for *Die Judenbuche* (J) and *Michael Kohlhaas* (K).

mentioned types (other, short and O+S) have an average length of 6.93 words (J) and 5.75 words (K). Of those matches, 45 (J) and 79 (K) are string equal when case is ignored. The average normalized Levenshtein distance of the source and target text string is 0.95 (J) and 0.92 (K). These results show that it is very likely that most of the false positives are not actually false positives.

5.3 Ablation Study

The presented approaches differ in the functionality they support as shown in Table 8. To evaluate the influence of the different functionalities, we compare the results of different versions of Lotte which emulate the absence of different functionality (cf. Table 9).

BLAST is optimized for fuzzy matching of OCRed text and allows for a high number of mismatched characters. This results in a high number of errors and makes it hard to link specific functionality to specific results. Therefore, BLAST will not

be considered in this comparison.

Lotte-(*base*) is Lotte with all five functionalities (cf. Table 8) disabled. This results in low recalls of 0.29 (J) and 0.26 (K). Lotte-(*OI*) is the base system with order independent matching enabled. This more than doubles the recalls to 0.64 (J) and 0.74 (K) and explains why TextMatcher has the lowest recall of all systems as it is the only system that does not support order-independent matching. SimT on the other hand only supports order-independent matching and achieves a rather high recall. Lotte-(*OI+otm*) is the OI-system with one-to-many matching added. This again improves recall significantly.

The last three systems Lotte-(*fuzzy*), Lotte-(*skip*), and Lotte-(*ellipsis*) are all based on Lotte-(OI+otm) with one functionality added. The improvement in recall for Lotte-(skip) explains the better performance of Copyfind over SimT.

Although around 16 % (J) and 19 % (K) (cf. Table 6) of quotations contain ellipses, the performance of Lotte-(ellipsis) is not a lot better. This might be because even without explicitly handling ellipses, a system will still find at least some part of the full match.

One more notable result is the high precision for all the different variants of Lotte. As discussed earlier, our data makes it hard to accurately evaluate the precision. We therefore decided to optimize for recall and assume a higher precision based on our findings in Section 5.2.

6 Visualizing and Exploring Key Passages

Here, we describe how the results of Lotte are integrated into an interactive website for visualizing and exploring key passages.[4]

6.1 Segmentation to Identify Key Passages

We process the output of Lotte to identify key passages by combining overlapping matches and generating minimal non-overlapping segments with frequency counts. Figure 2 sketches the segmentation process. The example contains the source text $w_1\ w_2\ \ldots\ w_9\ w_{10}$ and different sequences which quote the source to a varying extent. We segment the source text into non-overlapping segments and

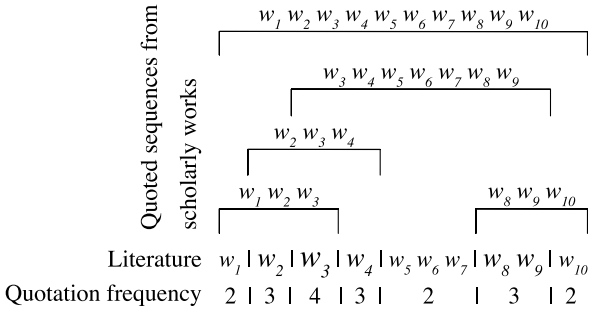

Figure 2: Visualization of the segmentation process.

count the frequency for each segment. For example, the sequence w_1 appears in two texts, sequence w_2 in three texts, sequence $w_5\ w_6\ w_7$ in two texts, and so on. This results in the quotation frequency shown at be bottom of Figure 2. The result of this segmentation process is used to visualize the literary text and the scholarly texts as described next.

6.2 Annette – A Visualization and Exploration Website

A screenshot of the website is shown in Figure 3. On the left, a heatmap of the complete literary text shows the distribution of quoted passages. The darker the text, the more often it has been quoted and thus the more important it is assumed to be. Next to the heatmap, the literary work is shown. The grayscale is determined by how many scholarly works quote some part of a key passage. That is, the color is always the same for the whole key passage. The font size is determined by how often a minimal segment is quoted. At the bottom, next to the literary text, a list of all scholarly works is shown. On the right, the top ten key passages are shown.

Starting from the initial screen, we can choose between different paths. The first option is to select a key passage by clicking on it. At the bottom, next to the literary text, a list of scholarly works which contribute to the selected key passage is then shown along with a preview of the quoted text. By clicking on one of the quoted texts, we can select a specific scholarly work. The text of that scholarly work is then shown at the top right. We can then go through that text and select other quoted passages. The bottom right shows how often the selected key passage was quoted and by how many scholarly works. Below, we can find the top ten most quoted segments of that passage. We can go back to the initial screen by clicking on the title at the top. From there, the other option is to select one of the

[4]The website is available at https://hu.berlin/annette-en. The source code of a white-label version is available at https://scm.cms.hu-berlin.de/schluesselstellen/lottevizex licensed under the Apache License 2.0.

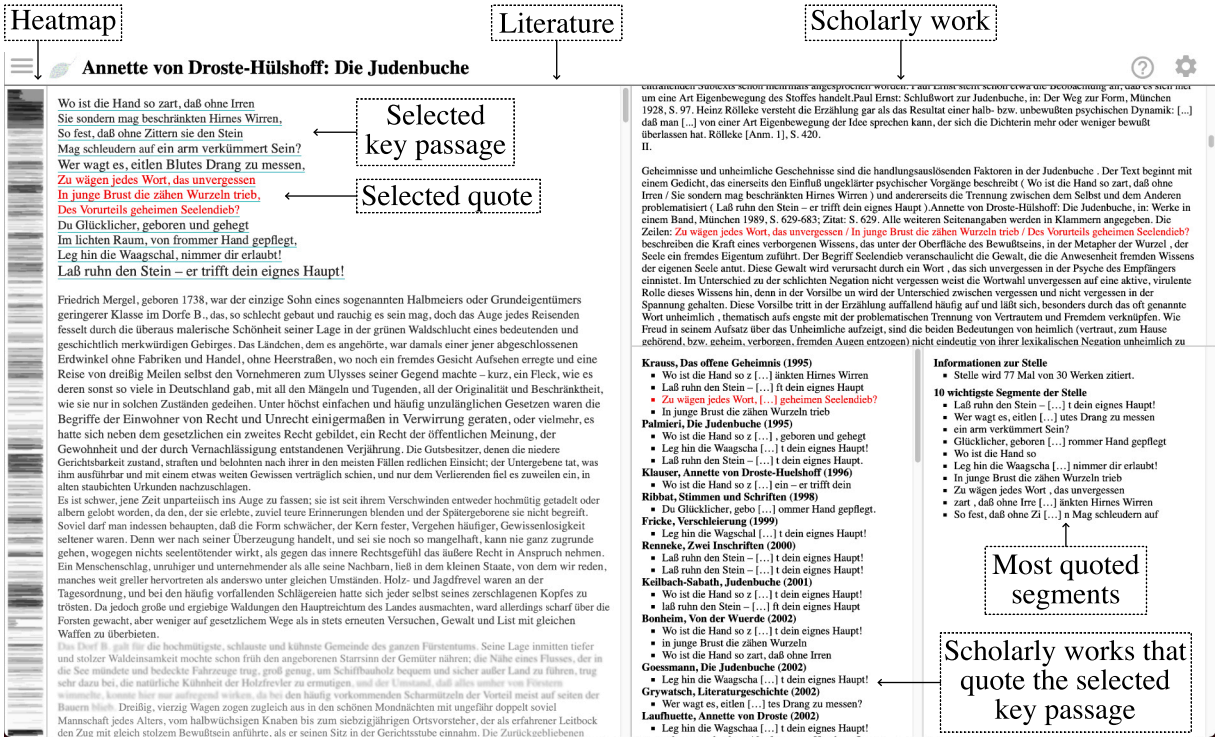

Figure 3: A screenshot of the website for visualizing and exploring key passages.

scholarly works from the list of all works. This will show the text of the selected work in the top right with all quotations highlighted.

7 Conclusion

We presented an approach for finding and visualizing key passages in literary works using scholarly works. For finding the quotations, we developed a system called Lotte by adapting Sim_text (Grune and Huntjens, 1989). Our approach outperforms prior approaches for text reuse detection. The matches are further processed to identify key passages by combining overlapping matches. We also presented Annette, a website that visualizes the literary work and scholarly articles together with the found quotations and thus allows us to explore the identified key passages and their origin. The current system only considers matches of length five and greater. In the future, we want to also identify shorter quotations and investigate how much information these add compared to longer ones. Another limitation of the current system is the missing suppport for handling ambiguous quotations. We have shown that this becomes more relevant the longer the source texts are. One solution to resolve such cases could be to utilize page references about the quoted passage, which are often included in the scholarly text. Furthermore, we aim to identify and analyze paraphrases and renarrations of literary works.

Acknowledgements

Parts of this research were funded by the German Research Foundation (DFG) priority programme (SPP) 2207 *Computational Literary Studies* project *What matters? Key passages in literary works* (grant no. 424207720).

References

Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. 1990. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410.

Lou Bloomfield. Copyfind [online].

Marco Büchler. TRACER: A text reuse detection machine [online]. 2016.

Annette von Droste-Hülshoff. 1979. *Die Judenbuche*. Insel Verlag, Frankfurt am Main.

Dick Grune and Matty Huntjens. Detecting copied submissions in computer science workshops [online]. 1989.

Timothy C. Hoad and Justin Zobel. 2003. Methods for identifying versioned and plagiarized documents. *J. Am. Soc. Inf. Sci. Technol.*, 54(3):203–215.

Sofia Kalaidopoulou. 2016. similarity texter: A text-comparison web tool based on the "simtext" algorithm. Bachelor's thesis, Hochschule für Technik und Wirtschaft, Berlin. Source code available at https://people.f4.htw-berlin.de/~weberwu/simtexter/app.html.

Heinrich von Kleist. 1978. Michael Kohlhaas. In Michael Holzinger, editor, *Werke und Briefe in vier Bänden*, pages 7–113. CreateSpace Independent Publishing Platform.

Vladimir I. Levenshtein. 1966. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10(8):707–710.

Derek Miller. To Quote or Not to Quote [online].

C. D. Paice. 1980. The automatic generation of literature abstracts: An approach based on the identification of self-indicating phrases. In *Proceedings of the 3rd Annual ACM Conference on Research and Development in Information Retrieval*, SIGIR '80, page 172–191, GBR. Butterworth & Co.

Jonathan Reeve. Jonathanreeve/text-matcher: First zenodo release [online]. 2020. version 0.1.6.

M. Scott and C. Tribble. 2006. *Textual Patterns: Key Words and Corpus Analysis in Language Education*. Studies in corpus linguistics. J. Benjamins.

Malcolm Slaney and Michael Casey. 2008. Locality-sensitive hashing for finding nearest neighbors. *IEEE Signal Processing Magazine*, 25(2):128–131.

David A. Smith, Ryan Cordell, Elizabeth Maddock Dillon, Nick Stramp, and John Wilkerson. 2014. Detecting and modeling local text reuse. In *Proceedings of the 14th ACM/IEEE-CS Joint Conference on Digital Libraries*, JCDL '14, page 183–192. IEEE Press.

TEI Consortium, eds. TEI P5: Guidelines for electronic text encoding and interchange [online].

Aleksi Vesanto, Filip Ginter, Hannu Salmi, Asko Nivala, and Tapio Salakoski. 2017a. A system for identifying and exploring text repetition in large historical document corpora. In *Proceedings of the 21st Nordic Conference on Computational Linguistics*, pages 330–333, Gothenburg, Sweden. Association for Computational Linguistics.

Aleksi Vesanto, Asko Nivala, Heli Rantala, Tapio Salakoski, Hannu Salmi, and Filip Ginter. 2017b. Applying BLAST to text reuse detection in Finnish newspapers and journals, 1771-1910. In *Proceedings of the NoDaLiDa 2017 Workshop on Processing Historical Language*, pages 54–58, Gothenburg. Linköping University Electronic Press.

Simone Winko. The making of plausibility in interpretive texts. Analyses of argumentative practices in literary studies [online]. 2017–2020. DFG-funded research project (grant no. 372804438).

Jin-ge Yao, Xiaojun Wan, and Jianguo Xiao. 2017. Recent advances in document summarization. *Knowledge and Information Systems*, 53(2):297–336.