CASINO TIMES
Compression and Similarity Indexing for Time Series

Master's Thesis of

Marco Neumann

at the Department of Informatics
Institute for Program Structures and Data Organization (IPD)

Reviewer:	Prof. Dr.-Ing. Klemens Böhm
Second reviewer:	Prof. Dr. Walter F. Tichy
Advisor:	Dr.-Ing. Martin Schäler

Submission Date:	2nd of August 2016
Presentation Date:	19th of August 2016

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, 2nd of August 2016

. . . . . .

(Marco Neumann)

Abstract

The detection of similarities withing the time series provided by the Google *n*-gram data can help researchers to explore and understand relationships between concrete words and abstract concepts. We construct a way of expressing this similarity and explain why this is, in our opinion, a sane approach.

Another important aspect of handling this kind of data in large scale the existence of an index structure for this task. We show how the prior art performs and why the time series data set is different from many other data sets. We then explore another way of exploiting similar information between time series to construct an index and compress the data at the same time. Afterwards, we show why our approach might have some essential issues and discuss some possible workarounds.

After presenting our way of implementing tools in this domain, we run a wide set of evaluations regarding the semantic meaning of our similarity metric, the effect of the compression on the query processing, the filter design that uses our system as an index and finally performance measurements. While the selected baseline provides sane result, it turns out that our approach suffers from the fact that greedy-decisions are only optimal in a local domain. In a global perspective they are often the wrong decisions and therefore we are unable to reach good compression rates while keeping the distortion low.

We end this work with some conclusions about the, not necessary great, results, and give an outlook on future research and what, in our opinion, might be other approaches worth to try.

Zusammenfassung

Datengetriebene Forschung ist, wenn auch nicht neu, ein essentieller Trend des 21. Jahrhunderts. Dies gilt nicht nur für Disziplinen wie Naturwissenschaften und Wirtschaft sondern zunehmend auch für die Gesellschaftswissenschaften. Ein wichtiger Datenbestand in diesem Sektor ist das Google *n*-Gram Projekt. Es bietet Einblicke darüber, wie sich die Verwendung von Wortketten (*n*-Grams) über die Jahre hinweg entwickelt hat, gemessen am Bestand aller bei Google Books erfassten Bücher. Dies ermöglicht es, Beziehungen zwischen konkreten Wörtern aber innerhalb und zwischen abstrakten Konzepten herzustellen.

Ein grundlegendes Problem dabei ist, dass die Überprüfung dieser Beziehungen bisher häufig auf Vermutungen basiert und damit den Effekten des Framings und des Confirmation Bias unterworfen sind. In dieser Arbeit wird deshalb ein neutrales Maß entwickelt, um derartige Links zu messen und objektiv zu bewerten. Diese Metrik, welche während der Konstruktion immer wieder auf ihre Bedeutung hin untersucht wird, basiert auf dem sogenannten Dynamic Time Warping (DTW) der Gradienten der Zeitreihen.

Nach der Herleitung einer Ähnlichkeitsmetrik besteht nun die Notwendigkeit, diese schnell evaluieren zu können um von einem technischen System in kurzer Zeit die Nachbarn, in Bezug zu der Metrik, erfragen zu können. Es wird gezeigt, warum bisherige Verfahren an den Besonderheiten dieses Datensatzes scheitern. Danach wird ein System entwickelt, dass die Ähnlichkeiten zwischen Zeitreihen ausnutzt, um gleichzeitig einen Index für die genannten Anfragen als auch eine Kompression der Daten zu ermöglichen. Dieser Ansatz basiert auf der Transformation der Daten in Haar-Wavelets und deren Darstellung als Baumstruktur. Die resultierenden Bäume werden dann in einem Greedy-Verfahren miteinander verschmolzen. Dieser Ansatz ist notwendig, da eine allumfassende Suche nach optimalen Vereinigungen eine zu hohe algorithmische Komplexität aufweist und deshalb unpraktikabel ist.

Es zeigt sich allerdings, dass das Greedy-Verfahren Schwächen hat und zu häufig lokal optimale aber global suboptimale Entscheidungen trifft. Es werden deshalb mehrere, nicht notwendigerweise erfolgreiche, Versuche unternommen, dieses Problem zu umgehen. Außerdem werden einige Alternativen vorgestellt, die stattdessen möglich sind.

Ein weiterer wichtiger Teil dieser Arbeit ist die performante Implementierung der Algorithmen. Dies wird in einem extra Kapitel erläutert.

Anschließend werden mehrere Test und Vergleiche bezüglich der semantischen Bedeutung des Ähnlichkeitsmaßes, der Qualität der komprimierten Daten, der Eignung des Ansatzes als Index-Struktur und der Performanz des Verfahrens im Vergleich zur naiven Bruteforce-Suche als auch einer bisher bekannten Index-Struktur durchgeführt.

Die Arbeit endet mit einer kurzen Zusammenfassung und Vorschlägen und Empfehlungen bezüglich weiterer Forschungsarbeiten auf diesem Gebiet.

Contents

List of Figures

List of Tables

List of Algorithms

1. Introduction

Books are, even in times of blogs and electronic publications[1], the most important way to publish fictional and non-fictional stories, reports, education material, well-collected scientific material and lifetime achievements. The writing within these books is material created at a certain point of history within a specific society under concrete, but not always known, circumstances. This is reflected in the usage of grammar and vocabulary. This reflection can be seen when reading a specific book, but it could also be seen in a statistical way when analyzing many books. A major barrier is that someone needs to read these books and at the same time is actively looking for specific patterns, until recently. Google as one of the big companies that handle data was able to collect the content of a variety of books within their Google Books[2] project. They then gathered the content of all books and collected statistical information about chains of words, called *n*-grams ([1]). The information is presented as time series, which store the usage count of specific *n*-grams over years. This enables people to check assumptions about how words were used over time and they can use this information to conclude about history of concrete phrases and abstract concepts. The data already helped other groups to extract valuable knowledge in linguistics ([2, 3, 4]), NLP for video descriptions ([5]), probability research ([6, 7]), investigation of cultural change ([8]) and to improve spell and grammar checking ([9]).

A major question that arises when analyzing single words or *n*-grams is if they are influenced by others or if there is any connection between particular groups of words and phrases. We have seen two important issues when someone's tries to answer this question. The first one is to find a scientific measure of "influence". Without such concrete way people will be affected by confirmation bias, which makes most of them rate what they see depending on what they expect ([10, 11]). This effect is even stronger when we do not provide them tools that present candidates simultaneously so a sequential checking is required ([12]). The second one is two come up with ideas which candidates could have a connection. People, no matter of which profession, have a framed, subjective view on the world and therefore are unlikely to start their research with unexpected result ([13, 14]). The existence of both issues hinders researchers to apply the scientific method to some fields and let them rely on expert knowledge and skills.

In this work, we provide a possible solution to the first and a partly complete solution to the second problem. We explore, discuss and define what "similar" means in our context and how a metric can be designed that matches this semantical idea. In addition to comparing entire time series, we also ensure that users can only take subranges of the series into account, so queries can be limited to certain epochs. This may not be the only

[1] This is 2016. It is likely that this will change.

[2] `https://books.google.com/`

way to describe similarity and we want to point out that it is important to check whether our proposal is suited for your work before applying it. Of course other fields of application may require small changes or totally different approaches on how to describe connections between time series. The similarity measure directly leads to a way of finding similar *n*-grams to a given query. The reason why this solves the framing problem only partly is that a user still needs to start with a query and because a naïve search for neighbors is too slow when using the complete data set that Google provides.

To solve the second issue completely we need to speed-up the lookup-process. This issue here is that, compared to other time series data sets, the Google *n*-gram data, which needs to be transformed in order to enable the similarity calculation, contains a lot of high-frequency information and therefore indexing the data is prevented by the curse of dimensionality. This makes the data set special when comparing to other time series, e.g. the ones considered in [15]. Therefore, we need to explore our own way on how to store the time series data in a way that exploits similarities between them. This is done by transforming the data and then look for parts that can be shared between the transformed instances. It leads to a compression effect and speeds up query processing in a way that enable others then simple single-*n*-gram queries. Also, this enable researcher with low computational power and storage[3] to process the whole data set of 1 to 5-grams.

Enabling fast search queries is not only an advantage for human users of the system. It also enables machines to gain information about the history of words and phrases and may enable novel techniques to visualize and analyze the data. The relationship of *n*-grams can then be expressed as pair-wise distance, as distance to carefully selected words, as graph with or without edge label or as multi-graph that takes time slices into account. The fast on-demand availability of the similarity data can be key to a whole new kind of research work and, without doubt, will lead to interesting results. To archive fast results within the constraints of the methods we choose, we will provide a state-of-the-art implementation which uses efficient memory management, clever pre-computation and modern CPU technology to provide results in the lowest possible time.

An important disclaimer: the *n*-gram corpus used for this work makes the assumption that every usage of an *n*-gram has the same weight. This was already criticized by other researchers ([16]) and we are aware of this problem. On the other hand, this data set is the only publicly accessible one of this kind and size and since there are not alternatives, we do not have a real choice. In principle our techniques also work with weighted sums, which may be available in the future. We hope that large scale literature research gets easier and we urge stakeholders to establish a workable fair use policy and to build up a digital archive that includes works from a wide variety of publishers and provides a proper API. Right now we are limited to bypass copyright limitations and rely on the, surely not selfless, kindness of big companies.

This work will first explore prior work and will set this contribution into context. Then we create a baseline by explaining data mangling and clean-up, which includes important decisions regarding *n*-gram normalization and information pruning. We then illustrate the choice of a similarity measure and why this leads to the curse of dimensionality, the reason why indexing this kind of data does not work with known methods. Afterwards,

[3] compared to companies like Google

we will present our method from its foundation to a concrete algorithm and optimizations as well as reasons why our original idea works worse than expected. Then, we will discuss implementation details and will run a full analysis and comparison with the baseline. At many points we present alternatives to our approach, which we encourage the reader to take serious. We conclude this work with possible future research topics and an outlook on the benefits of data-driven research based on this type of data.

2. Prior Work

Before starting with the presentation of our research we would like to introduce some prior work and discuss how they differ from our work. First, we want to emphasize the following point, which we think it is not expressed enough amongst researchers: these methods and algorithms are not bad in general and have their applications, but there are reasons for not using them. Either the time series we work with are just different from the ones that are normally used or it is because many fields do not face the vast amount of data as we do. And some of them are just not able to simultaneously match the goals we have:

- finding similarities as defined by us amongst a large set of short but entropy-right time series
- use these findings to speed up nearest neighbor queries
- at the same time exploit shared information to compress the data
- enable similarity search on subranges of the data set with different sizes without the need to prepare the data again
- while the preprocessing is done once and can be slow, the nearest neighbor queries need to fast enough to enable interactive investigations
- the entire data processing must be possible on one server to enable more groups to work with the data

2.1. Dimension Reduction

The first idea on how to deal with our data set would be a dimension reduction. To do so, the time series data is usually transformed into another representation and from that multiple parts are removed. This can be either done as a compression technique or to speed up index lookups ([15, 17, 18, 19]). As shown in the previous section, this prunes too much information to enable good indexing. Therefore, our main goal is to not reduce the dimensionality of the data but rather exploit similarities of the time series to compress data and build an index to speed-up DTW calculations.

2.2. No Compression

Some methods ([20, 21]) try to transform the data into another representation and use this representation to speed up similarity search. These transformations are only used for

indexing purposes and do not compress the overall data set. This does not lead to a data reduction since you still need the full data corpus, at least when checking for our definition of similarity, and therefore does not enable users to process the full *n*-gram corpus. Since the preprocessed 1-gram data set is 1.6 GB and the 2-gram set is already 13 GB in size, we expect that the full 5-gram data set can only be used for interactive query processing when compression is used.

2.3. Feature Extraction

Another whole topic is the extraction of specific features ([22, 23, 24, 25, 26]). It leads to a compression and sometimes ([22]) to human-readable outputs. The reduced and refined amount of data usually enables better index structures. The reason this is not applicable to our problem is the following: during the feature extraction process it is not clear how large or small the time ranges during the similarity search will be and therefore it is not clear how fine-grained the resulting description should be. We have shown that the frequency of our time series is high and a coarse-grained extraction would be equal to a dimension reduction and therefore would not work as well. A fine-grained extraction results in too much information to index and process on demand and is also a bad compression.

For small fixed query windows a feature extraction might work but keep in mind that the neighborhood search still needs to match our similarity definition and therefore must be able to compensate small warping effects. Keeping the curse of dimensionality in mind the number of extracted features should possibly not exceed a count of 16. Together with the frequency plots shown in Figure 3.11 we guess that it might be possible to enable meaningful feature extraction for time windows up to a size of $16 \cdot 256/64 = 64$. From our own experiments in subsection 3.2.4 we think that this approach would also lead to large changes within the set of nearest neighbors. There might be features that can archive this task or even be able to exceed our guessed window but we are not aware of any existing publications covering this.

2.4. Similarity

It is not surprising that others also tried to find similarities amongst time series.

One example is similarity based on mutual information ([27]). We believe that this definition is a great idea and is a very generic approach, which can be applied to many fields. Sadly their method relies on slow pairwise comparison. Also, their idea of relationship does not match the one we developed for this particular application.

The definition developed in [28] is based on the matching of rescaled subsequences. This is similar to our idea of using DTW and might be worth to consider for future research work. For now we are not able to use this approach because it does not scale well enough to the large amount of time series.

An interesting approach is presented in [29]. In that paper time series are described by rules that describe discretized data. In our opinion that could be classified as feature extraction. The two issues are: the discretization loss and the amount of extracted rules.

Both do not work very well with our idea of similarity, but it might be worth to consider to use this technology to add additional information to behavior that we find with our method and make the results more understandable to the users.

One last idea, which we want present here, is an alternative to the used DTW: Longest Common Subsequence (LCSS) and Edit Distance on Real Sequence (EDR). These require either discrete data or ϵ-thresholds. [30] crafts a generic and fast approach to compute these. Other groups might find this helpful but we decided that DTW is better suited for our application since its idea of a Euclidean distance with some time inaccuracies better match our model of similarity. For us that means not including binary decisions during the distance calculation.[1] The reason for this is that the difference of the distance within the set of nearest neighbors is small as shown in subsection 3.2.4.

2.5. Related Techniques

Noise reduction techniques ([31]) are not desired in our case since we already know how to remove noise properly and that the level of noise reduction depends on the query behavior of our users.

[1] Strictly speaking the DTW calculation does include binary decisions but these are only be used to find the minimal distance not for the actual distance calculation.

3. Baseline

Before we can start analyzing any data or to do any algorithm research, we need to clarify what exact data we want to process and what results we want to gather. In this chapter we design the preprocessing of the data from the collection from the Google server over normalization up to the clean up. Then we craft a precise description of the what we think similarity is in our context.

3.1. Data

As for most data analysis tasks, the raw input data is not suitable for direct information gathering. To clean up the data we are looking for an efficient and deterministic way. To do so we perform the following clean up steps:

1. download and parsing: Retrieve the data from the Google servers and parse the data for the first time. This step results in a list of all n-grams within the data set but does not, for efficiency reasons, store the actual time series data.

2. string filtering: Remove strings from the list of n-grams that contain certain characters. This removes punctuation and other unwanted entries.

3. string normalization: At this point we handle the fact that semantically identical Unicode strings might be encoded in different ways.

4. word normalization: For our application we ignore different word attributes like tenses and pluralization. This results in an unambiguous list of n-grams.

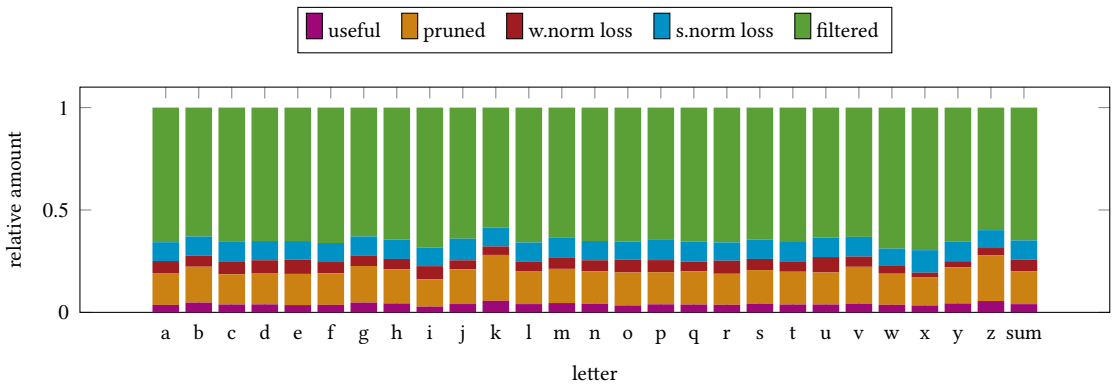

Figure 3.1.: Relative amount of pruned data per letter

5. pruning: the Google data set contains a lot of very rare entries that do not contain enough valuable information. At this point, we strip certain parts based on a time frame as well as statistical significance. This is the first time that the actual time series data is required. This delayed parsing removes a lot of overhead during the early stages of the pipeline.

Note that swapping the string-based steps does not lead a different output. Figure 3.1 show how much data is pruned by every step for the 1-gram data set. The exact numbers can be found in Table A.1 and Table A.2. Naturally there are some differences between the different starting characters of the *n*-grams but as far as we can tell no anomalies can be observed.

If not stated otherwise, all figures and explanations in this work are derived from the 1-gram data set.

3.1.1. Download and Parsing

As a first step we download the raw data from the Google server. A complete list of download URLs can be obtained here:

`https://storage.googleapis.com/books/ngrams/books/datasetsv2.html`

We use the files of version `20120701` listed under "American English", with the exception of:

- files with *n*-grams that for sure contain word types instead of types and therefore will be dropped[1]: `_ADJ_`, `_ADP_`, `_ADV_`, `_CONJ_`, `_DET_`, `_NOUN_`, `_NUM_`, `_PRON_`, `_PRT_`, `_VERB_`
- files containing *n*-grams with numbers: `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`
- punctuation data: `punctuation`
- characters which do not belong to the English alphabet: `other`

The data collection and processing is described in [1]. The raw text data, which was downloaded from the Google servers, is mangled by a high-performance C++ implementation to speed-up the upcoming steps. During the first steps, only the *n*-gram content is required without any knowledge of the actual time series data. We exploit this fact and delay the actual parsing and storage of the time series until the data is required and focus on the handling of the string data as long as possible.

3.1.2. String filtering

It seems that the *n*-grams that Google extracted does not only contain pure words but also numbers, punctuation characters and word classes. To simplify storage of the strings and because our users are not interested in searching for whole word types, we filter out all entries that contain characters of the following class:

[1] Does not apply to the 1-gram data set.

$$\_./, :; ! ?\backslash'"\#() <>= + * \{\}0 - 9$$

We have found this character set sufficient for our work, but want to point out that there this is a parameter that depends on the concrete application. For example, we drop *n*-grams that contain the combined words like "It's". To deal with this type of content, the pipeline needs to be extended to split words into a normalized form. In the following work we assume that all parts of the *n*-gram are atomic words.

3.1.3. String Normalization

Since the data the *n*-grams are described by Unicode strings, there may exist string describing the same content. We apply NFKC normalization as described in [32] to solve this issue. Furthermore we lowercase all inputs with respect to the Unicode standard. If this procedure will lead to duplicate *n*-grams, they are joined by adding the corresponding time series data.

Notice, that the lowercase transformation may not work as expected for other languages than English since it may lead to a loss of important information.

3.1.4. Word Normalization

The words forming the *n*-grams exist in multiple variants, e.g., different terms for verbs or singular and plural form for nouns. Naturally the resulting time series are very similar and do not contain valuable information, neither for our algorithms nor for a human analyst. We solve this issue by applying the WordNet lemmatizer ([33]) and afterwards the snowball stemmer ([34]) to all words. In case of same output *n*-grams the related time series are added again.

As for the former transformation step, be aware of the language problem. Other languages may require other word normalization techniques. Also, there may be more advanced techniques that exploit the knowledge of the entire *n*-gram instead of single words.

We decided not to run any OCR (optical character recognition) error recognition since we are not aware of any general purpose approach that does not transform rare words or names like a naïve spell checker would do.

3.1.5. Pruning

The original data sets contains a lot of very rare *n*-grams that, in our opinion, do not provide enough statistical information to be considered during the further analysis. The same applies to all *n*-grams for the early years that are contained in the data.

To handle the first case, we have decided to drop all *n*-grams where the related match count time series is too small, or in mathematical terms where $\sum_{1753 \leq y \leq 2008} v_{0,y} < 1000$ (v_0 is the "match count") applies. This threshold is, as many other parameters, an application-specific one. For us it was a trade-off of eliminating noise and keeping some very specific entries which might be required to answer some questions in the field of philosophy.

To eliminate the second source of irrelevant information we decided to only use the last 256 years of the time series data. The selection of the time range of 256 years has other advantages apart from the pure pruning. Since it is a power of 2 it is easier to apply many transformations (e.g., Fourier or Discrete Wavelet Transformation) to it without the need to think about and justify additional edge case handling.

Please note that this is the first step where the actual time series content is required. Therefore, it is also the first step where we parse the time series data for all *n*-gram strings that survived up to this point.

3.2. Similarity

The idea of similarity depends on the concrete application. The simplest one, which is obviously not suitable, is to just take the total count of *n*-grams over a time period and calculate the absolute distance to each other. Another idea would be to treat the time series as a high-dimensional vector and use the Euclidean distance as a description of how similar the different series are. It was not clear beforehand which precise mathematical definition is suitable for our application. It should be justified by two parts. The first one is a purely data-driven approach. Because we want to overcome the subjective guessing described in the introduction, we want a similarity metric which is based on the actual time series data and nothing else. The second part is that the results should be sane in a semantically sense. So for *n*-grams that are similar there should, from a user point of view, also be a good reason to trait them as such.

Notice that similarity is a function which describes a metric while not being mathematically correct. We guarantee non-negativity and symmetry but the method we come up with does not provide identity of indiscernible and subadditivity.

In this section we first discuss, which time series we want to consider as similar. We want to explore, which *n*-grams are used together. Because there might be a normal usage of certain words or phrases in the day to day life, interesting data might be hidden by an overall large number and therefore interesting changes in usage patterns might not be discovered when total numbers of mentions are compared. Consider the following example: $n_1 \Rightarrow n_2$ but not vice versa and n_2 is already used in many books but n_1 is a new *n*-gram at a certain point of time. This implication cannot be found by comparing total numbers, because n_2 might be way larger than n_1 but the implication will lead to similar changes in the overall time series. So we seek to find similar structure instead of similar usage numbers.

Definition 1 (Query)**.** *A query is a time series for which the user (or some program) wants to know the nearest neighbors for. By writing "some word" we refer to the time series that belongs to the normalized n-gram or the n-gram itself, depending on the context. The plural form queries is used to express multiple starting points for a nearest neighbor search, either when generating statistics or when calculating the distance to multiple time series.*

Furthermore we limit our analysis to the "match count" because it seems to be more useful for later research than the less fine grained "unique book count".

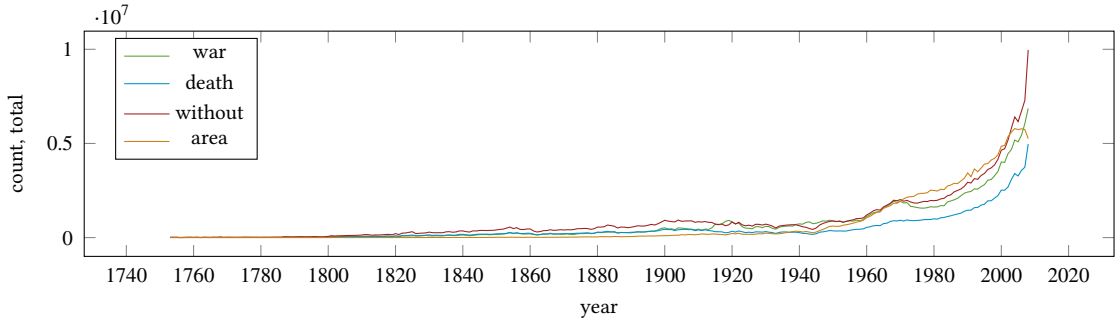

Figure 3.2.: Total counts of example *n*-grams

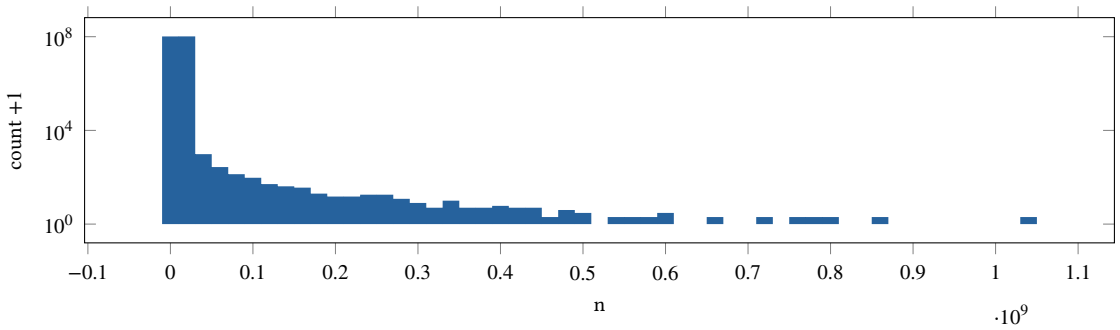

Figure 3.3.: Histogram of all time series

To derive meaningful similarity, we will now explain the reason and execution of the different steps done for the metric calculation, which are executed in the following order:

1. normalization: since the book counts grow exponentially but at the same time we want to ensure independent preprocessing of the single time series, we do a $\log(x+1)$ transformation
2. smoothing: to eliminate noise for the next step[2]
3. gradients: we seek for structural changes withing the time series, not similar counts, so we calculate the distances of gradients instead of the original time series
4. Dynamic Time Warping: time series can include small delays we want to compensate them before doing the actual distance calculation
5. ranking: similarity can be used to derive a set of nearest neighbors; we explain how to do so, which problems occur due to noise and present one possible way to automatically select the size of the set of nearest neighbors

3.2.1. Normalization

In Figure 3.2 you can observe the following: the words “without”, “area” and “death” share the global low around 1945 while the word “war” does not. On the other hand you can

[2]This step does have its own subsection but is explained during the discussion of the gradient calculation.

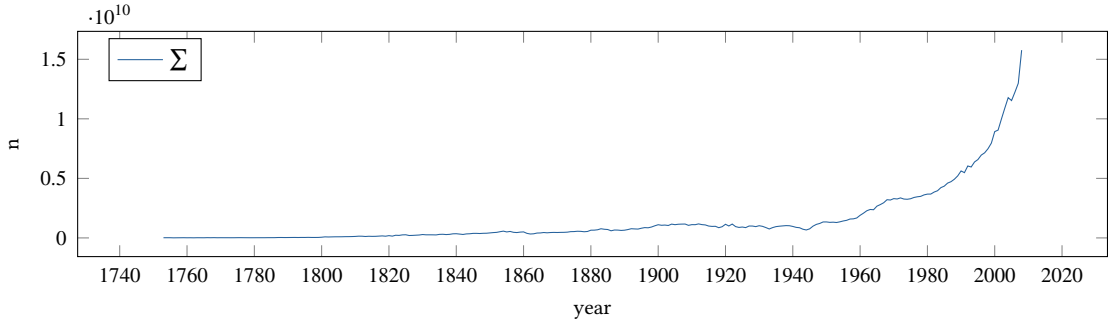

Figure 3.4.: Absolute sum of all n-gram time series

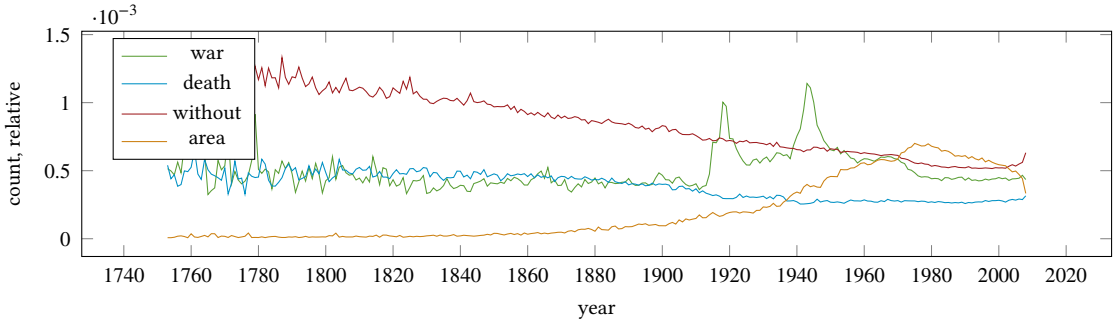

Figure 3.5.: Relative counts of example n-grams

see the same structure (peak and then depression) between 1961 and 1977 for “war” and “without”. Another feature that can be observed is the fact that the time series data is not limited in its value and that they behave exponentially, as shown in Figure 3.3. Exponential growth is also common amongst this area and our natural environments ([35, 36]). To be able to determine a proper similarity between different time series, this scaling should be eliminated. There are two possible ways: applying a logarithm or normalizing every point in time by using a factor shared amongst all series.

Figure 3.4 shows the foundation of such a factor. The problem with this approach is that time series with an overall huge impact on the sum will also influence the normalized results of small time series heavily. In other words: rather than symbolizing a straight growth, the sum itself has a structure, which then will influence the normalized result. An example result of this normalization is shown in Figure 3.5. It can be observed that the n-gram “war” seems to grow heavily during the time around the world wars. The reason for this is that the overall number of publications decreased during this time except for war-related topics. As explained, the structure of the time series sum itself now results in a new feature of the “war” data.

So we choose to apply $\log(x + 1)$ to all values for normalizing. Note the 1 that was added because the range of possible input values starts at 0. The results are shown in Figure 3.6 and the corresponding histogram is shown in Figure 3.7. There is a small gap in the histogram, which is the bin $(0, 0.5]$, which before transformation is $(0, 0.65]$ and therefore no count values can fall into that range. Overall the results look way better distributed. It can also be observed that the later years are smoother than the beginning

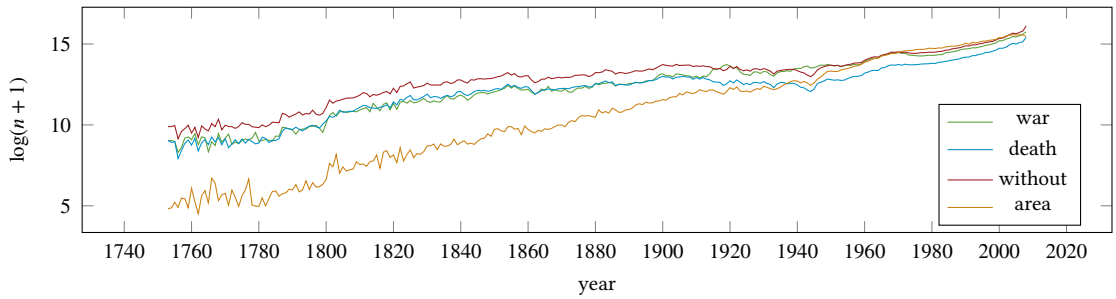

Figure 3.6.: log counts of example n-grams

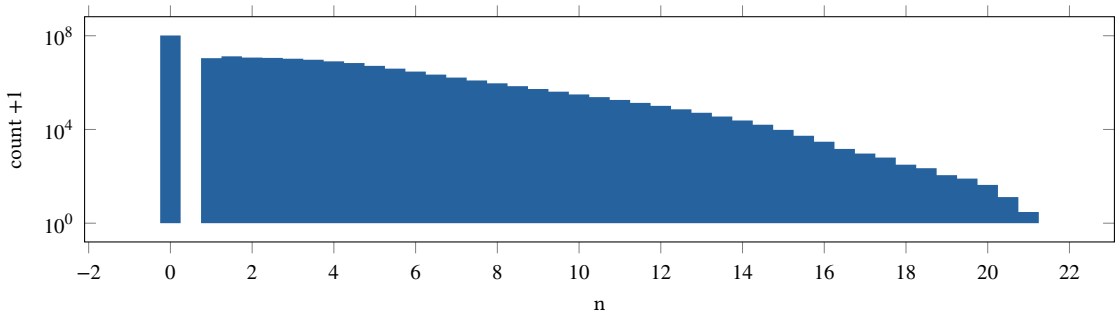

Figure 3.7.: Histogram of $\log(x + 1)$ of all time series

of the time series. This is due to the fact that larger values counts are, in relative means, less noisy than smaller counts.

An alternative to a normalization could be a equal-frequency binning. This would remove any monotonic transformation function from the input data and equalize the histogram. These advantages come with a major drawbacks, which are the reason why decided against this approach: In the following chapter, we will introduce the need for a gradient calculation. To do so, the bins must be chosen very fine-grained which on the other hand introduces additional noisy to the data. To overcome this problem someone could use a coarse-grained, equal-frequency binning to guess a transformation function, e.g. based on a polynomial in general or a cubic spline in particular, similar to the approach presented in [37]. We did not implement this approach but it might be worth to test it during further research.

3.2.2. Gradients

As Figure 3.8 shows, there are cases where time series look very similar but are difficult to fit by using a linear transformation. Therefore, a simple p-distance is not sufficient for as a distance measure. We decided to fit the gradients instead. An example is shown in Figure 3.9. We use the $\Delta = 1$ gradient of $\log(x + 1)$. Now time series with similar structure have a low distance. The following two subsections will explain two additional tunings we made to make this distance more meaningful.

Since calculating the distances of noisy gradients may lead to useless results, we smooth the time series using a Gauss kernel with $\sigma = 1$ before deriving the gradient. An example

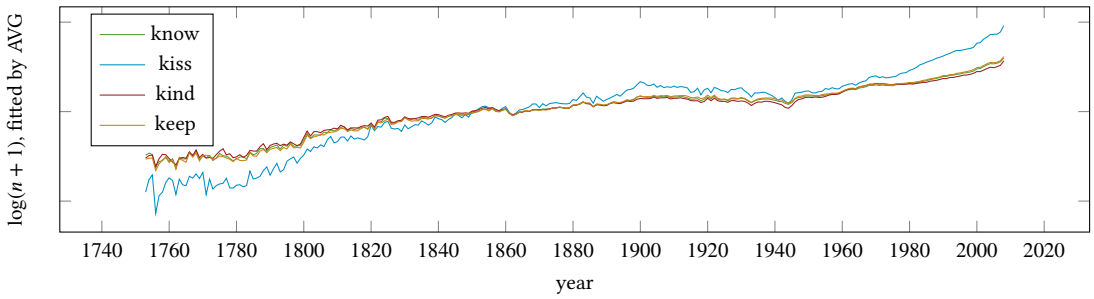

Figure 3.8.: Example *n*-grams, fit using AVG

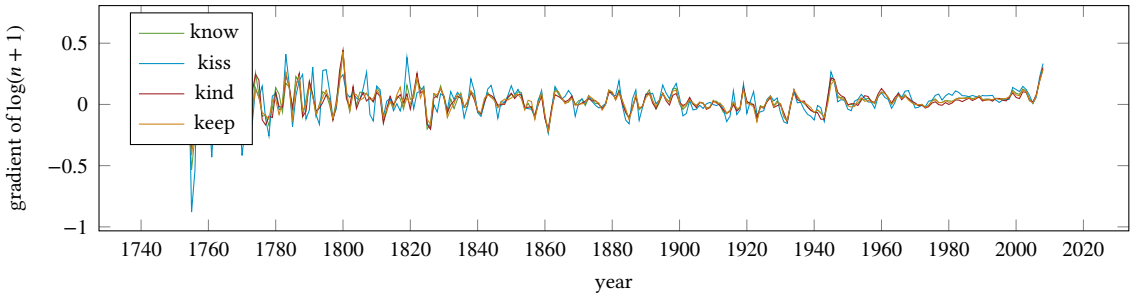

Figure 3.9.: Gradients of example *n*-grams

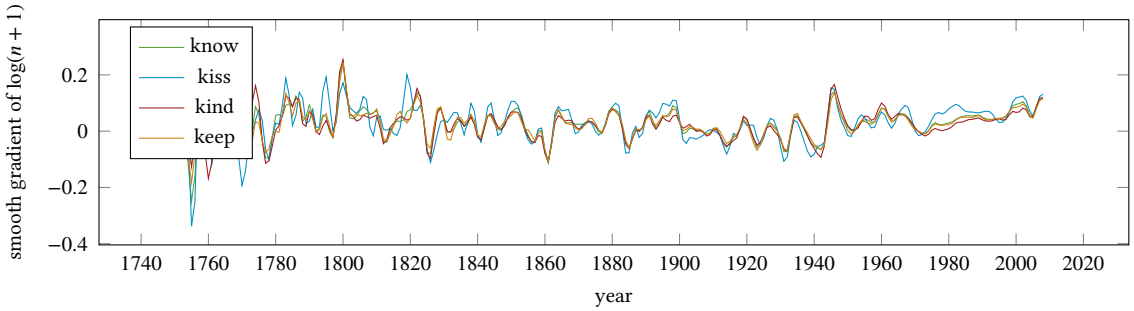

Figure 3.10.: Gradients of example *n*-grams, smoothed with $\sigma = 1$

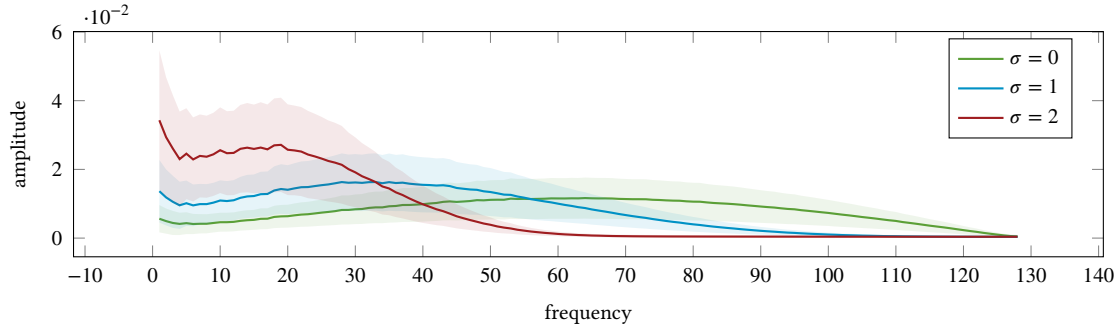

Figure 3.11.: Frequency distribution (mean + standard deviation) for different smoothing levels

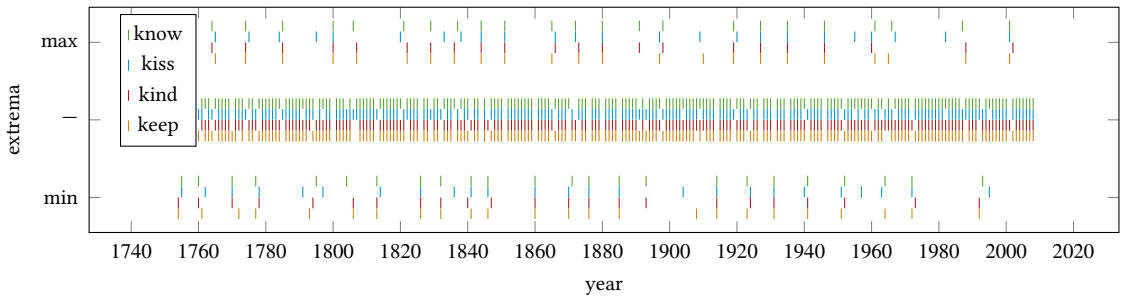

Figure 3.12.: Extrema of gradients of $\sigma = 2$ smoothed example time series

is shown in Figure 3.10. The reason we can use a Gauss kernel, which takes past and future values into account, for a time series here is the fact that the data we are transforming is known beforehand. This is different to most other scenarios where smoothing techniques like EWMA (exponentially weighted moving average) are required. Frequency distributions for different smoothing levels are shown in Figure 3.11. The non-smoothed data shows quite high amplitudes in higher frequencies ranges, which is uncommon for most time series data sets. The chosen smoothing of $\sigma = 1$ reduces noise while $\sigma = 2$ leads to a heavy loss of information. The overall pattern of fast switching gradients will also lead to some other problems, e.g. in section 3.3. The similarity of the 1-gram "kiss" is clearer now. As expected, the time data does not quite fit the other example series. This is intended since the underlying data also has a slightly different structure.

Instead of smoothing the data before calculating the gradient, there are other ways of comparing the derivative of the time series while avoiding noise problems. For example, the time series could be approximated by a polynomial from which the derivative is used as an input for the DTW calculation. This would automatically reduce the noise of the input data. Since the derivative has a polynomial representation as well, this might also speed up further processing, like the DTW calculation ([38, 39]), which is introduced in the next subsection. Also, the transformation into a polynomial would lead to a compression effect. The drawback is that someone needs to find, either statically or dynamically, the right parameter for the degree of the polynomial as well as a proper algorithm for guessing the coefficients.

3.2.3. Dynamic Time Warping

Now we have reached a state where we could just calculate the quadratic distance of the smoothed time series data. The question is if this obvious approach is sane. If would do so, we would completely ignore the fact that events that influence one time series have an delayed effect on others. Someone might wonder if this effect occurs in our data set. Take a look at Figure 3.12 where we plot the extrema points of the smoothed data. While all four time series have the same structure, their smoothed gradients do not align perfectly. To calculate a meaningful distance, we use DTW (Dynamic Time Warping) as described in [40] with an Euclidean distance. The usage of gradients as an input of Dynamic Time Warping follows the idea presented in [41]. To only allow a warping up to a certain distance we use Sakoe-Chiba Band from [42] of radius r. This prevents the DTW

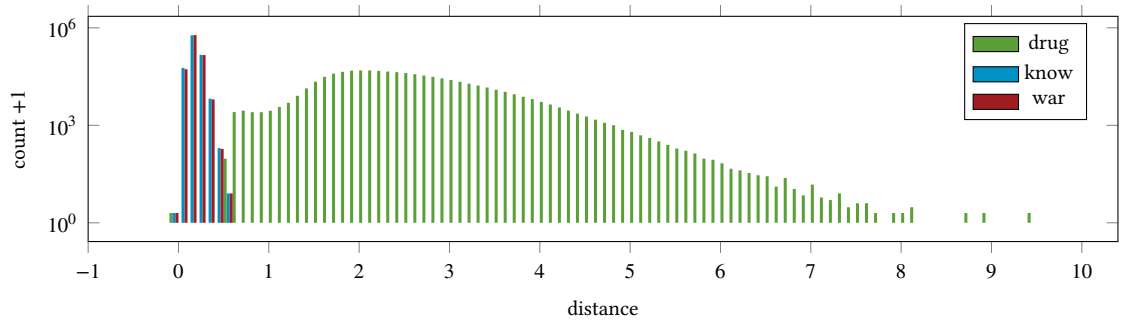

Figure 3.13.: Histogram of distances to example 1-grams

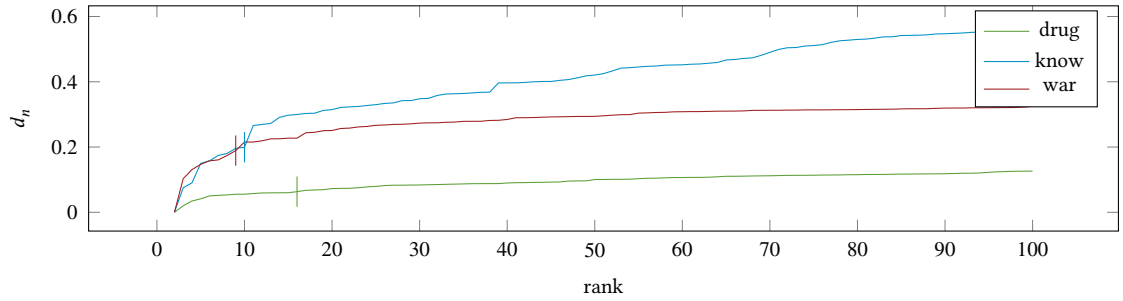

Figure 3.14.: Distance of the first 100 neighbors of example 1-grams, without the first element, which is the query itself

algorithm from stretching the time series too heavily which would result in semantically insane results and therefore would contradict our goals we have for our similarity.

3.2.4. Ranking

It seems obvious to use the DTW distances to rank nearest neighbors. There are two major problems with that approach.

First the distribution of the distances differs heavily from query to query as shown in Figure 3.13.

Second it is not clear how many nearest neighbors should be taken into account, especially when designing user-facing applications. To solve this problem we sort all neighbors by their distance and only take rank 1 to $\frac{m}{2}$ into account, with m being the size of the set of all n-grams. This ignores the query itself with its distance of 0 and prunes the second half of the sorted list because there are usually many outliers at the end of the spectrum. Then we normalize the distances:

$$d_{n,i} = \frac{d_i - d_1}{d_1} \tag{3.1}$$

Here d_i is the distance of the neighbor with rank i, starting at 0 with the query itself. The result is shown in Figure 3.14. Now we search for the index that clearly cuts the group of the nearest neighbors from the rest of data. To do so we take into account the following observation: the derivative of the function of sorted nearest neighbors is usually

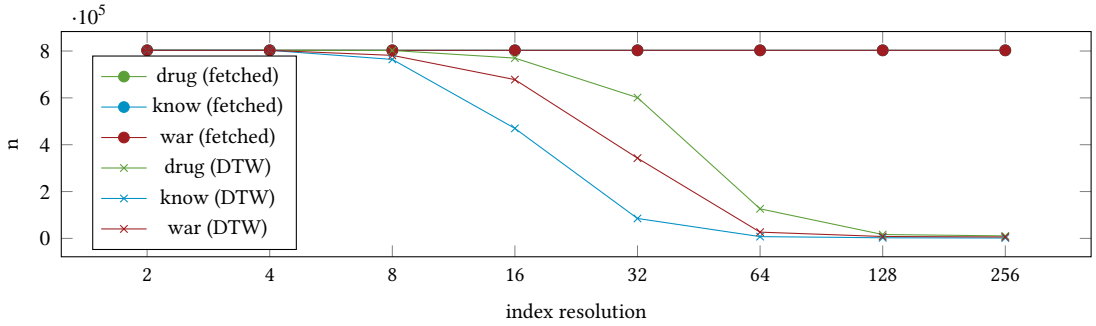

Figure 3.15.: DTW index efficiency, per resolution, r is 10

decreasing and is then suddenly increasing again, isolating a dedicated group of nearest neighbors. We use this as a cutoff-point. The mathematical term is:

$$i_{\text{cut}} = 1 + \arg\max_{i=1}^{\frac{m}{2}} \left(\Delta_1 \left(\frac{\Delta_1(d_n)}{d_n} \right) \right)_i \tag{3.2}$$

Here Δ_1 calculates the gradient with distance 1. Notice that we normalize the first derivative since we want to compare relative changes from a certain neighbor to the next one. This normalization is always possible due to the sorting of neighbors by rank. The second derivative can be positive and negative and therefore normalization is not desired and also would not have a semantical meaning.

That approach might seem to be slightly artificial and indeed its a more phenomenological metric instead of a mathematically proven approach. The main reason for this is that we want to keep the index as small as possible to enable users to understand the content of the list. If you try to use the set of nearest neighbors as an input for another algorithm you may want to choose another method.

Keep in mind that our method requires sorting of all neighbors while naïve k nearest neighbors search requires only partial sorting. For a bruteforce search this should not be a problem since calculating the DTW distances dominates the execution time but it might be a problem when handling overly large data sets. Also, it renders index techniques useless since the algorithm cannot ensure to draw enough data from the index to reach the cutoff point. Therefore, we do not use this technique during all evaluations.

We also want to point out that this method is very instable. Small changes in the input data, e.g. by noise introduces by compression, will quickly lead to different results.

3.3. Index-based Speed-up

One of the fundamental problems of large-scale DTW-based similarity searches is that, without further preparation, a linear scan is required. In preparation of further research and usage of 5-grams, we want to avoid this and cut the runtime complexity to $\mathcal{O}(\log n)$. While doing so we still want to archive exact results, at least for our baseline. We decided to implement [15], which is a combination of an R-tree index ([43]) and early sorting and pruning by utilizing bound checks.

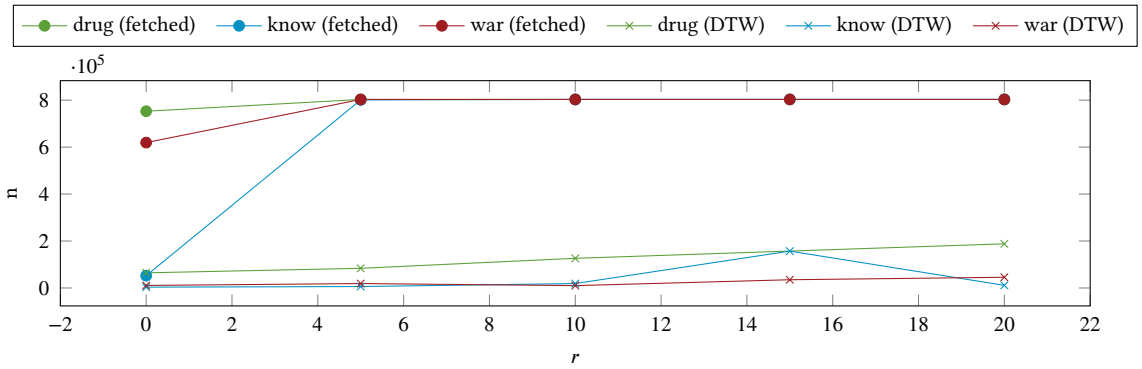

Figure 3.16.: DTW index efficiency, per r, resolution is 64

A problem with this type of index is that the structure of the indexed data is different than the one tested in the publication. Our rather short data (in terms of measured points) contains more entropy and is more jittering than the time series intended by the paper. Therefore, we require more dimensions to get a tree that at least filters out some candidates as shown in Figure 3.15. We need at least 16 dimensions to get any effect from the `LB_PAA` and we do not even see any real filtering effect from the tree structure itself[3]. So the tree structure only provides a sorting of the time series. That results in a very memory-inefficient data structure. Remember that for every dimension, the R-tree stores two data points, a minimum and a maximum. So we need to store at least 12.5 % of the actual data size as an index to get a any effect and still have a linear complexity. The high dimensionality renders the tree performance and memory management nearly useless, as also found by [44]. The situation depends on the warping radius as shown in Figure 3.16. For non-trivial values nearly the entire data is fetched from the index. Because all of this is a general problem of R-tree-based algorithms, we did not implement the improvements suggested by [17].

3.4. Alternatives

We want to point out that the baseline we have selected is not the only possible one. Other teams may come up with other setups depending on their definition of "similarity". Our choices depend on the application we want to use the algorithms for and understanding of the humans operating these applications. But even with the exact same requirements it may still make sense to choose another baseline. It is important to keep in mind that the choices made during this process may lead to different results and that this might affect research outcomes of people using the similarity results to explore the data and to justify the results of their analysis.

[3] the amount of received time series is more than 95 %

4. The Design of CASINO TIMES

In this chapter we explain our approach to compress the data and how we want to exploit this compression to find similar time series. To do so we first describe how we transform time series into tree structures, then we explain the general idea of information pruning. Afterwards we develop a method that instead of pruning of information relies on the merging of similar information packages. We then discuss different error metrics, detail how our construction can be used as an index and explain some technical improvements made to the algorithm. Finally, we show, which issues our approach has and how we tried to work-around these.

Before we start we want to define some important terms:

Definition 2 (Compression rate)**.** *The compression rate specifies, how much data is left after a certain compression is applied to the data. A higher compression rate is expressed by a smaller number and therefore signals a better compression effect. Memory usage is more efficient on higher rates. The compression rate does neither make any statements about resource usage during compression or decompression nor, in case of lossy compression, on the quality of the decompressed data or in other words: about the compression error and artifacts.*

4.1. Discrete Wavelet Transform

First of all, we need to find a time series representation that enables us to compare different series on different time scales and different granularities. While the granularity requirement is met by a Fourier Transform or a Discrete Cosine Transform ([45, 46]), it is insufficient for different time ranges. The reason for it is that the entire series is disassembled into frequencies that span the whole range. That makes it difficult to compare sub-ranges. A possible workaround would be the splitting of the time series into fixed-size chunks and transform each of them individually. Sadly, this makes it hard to handle requests for different time spawns and introduces an artificial choice between fine or coarse granularity, but never allows to extract and combine information for the entire range of granularity choices.

So we chose a different transformation — the Discrete Wavelet Transform under usage of the Haar Wavelet ([47]). A special property of this kind of transform is that it deconstructs a time series into a tree-like structure. Note that we only implement indexing of time series with length that is a power of two, because of our recursive tree-construction which requires subtrees of equals sizes. A generalization might be archived by the application of [48]. This limitation does not apply to query ranges.

To explain how the tree decomposition works we first need to clarify some terms:

Definition 3 (Superroot)**.** *Since in a wavelet tree every node only describes the difference between the left and the right subtree, the information about the absolute position of the time*

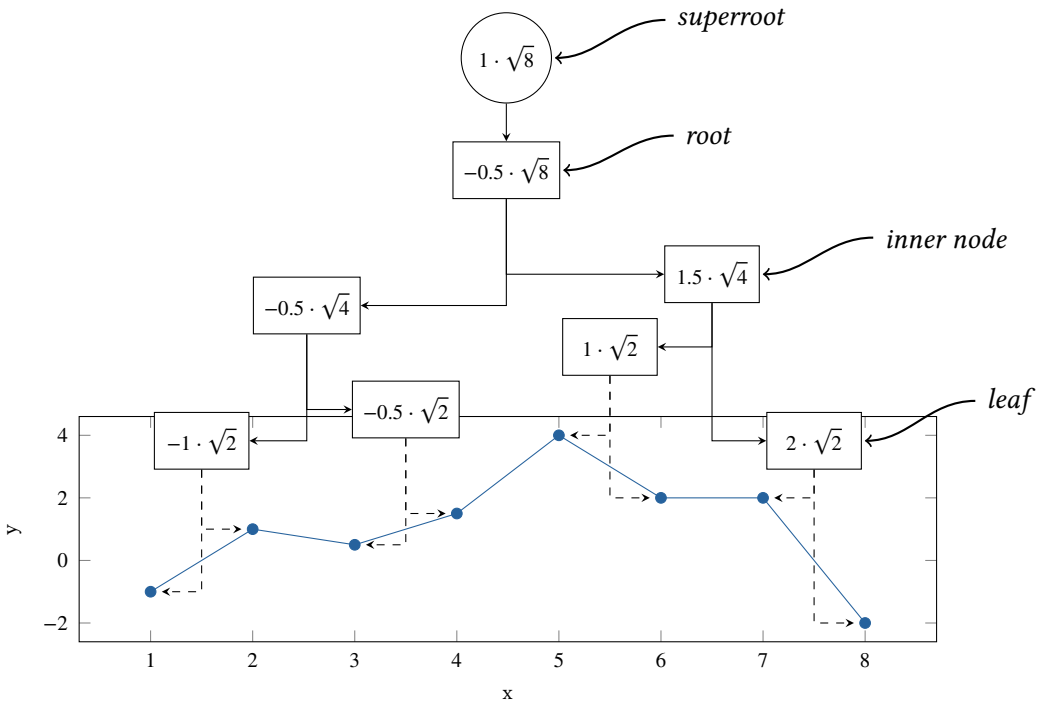

Figure 4.1.: Wavelet tree construction

series is missing. To store this information we introduce a superroot, a node that is the parent of the root and the only one that has exactly one child. This absolute position of the time series stored in that node is called anchor and has the symbol a. Since the wavelet trees are fully balanced the anchor equals the mean value of a time series.

Definition 4 (Depth, Level)**.** *A level is a node attribute measuring its distance from the superroot of the tree starting at* 1*. So the root of the tree corresponds to level* 1*.*

The depth of a tree defines how many levels it has. It can be calculated from the time series using the following formula:

$$d = \log_2 m \tag{4.1}$$

Now the tree construction s at the point in time t can be described with:

$$s(t) = 2^{\frac{d}{2}} a + N_{1,1}(t) \tag{4.2}$$

Here $N_{l,i}(t)$ describes the influence of a certain node in level l and delta i (since there are usually more than one nodes per level). The influence can be calculated with:

$$N_{l,i}(t) = \begin{cases} \mathbb{1}_{t \in N_{l,i}} 2^{\frac{d-l+1}{2}} x_{l,i} + N_{l+1,2i}(t) + N_{l+1,2i+1}(t), & l \leq d \\ 0, & l > d \end{cases} \tag{4.3}$$

Here $x_{l,i}$ is the coefficient stored in the node $N_{l,i}$ and $\mathbb{1}_{t \in N_{l,i}}$ has an effect to this point in time:

$$\mathbb{1}_{t \in N_{l,i}} = \begin{cases} 1, & t \in 2^{d-l} \cdot (2i-2, 2i-1] \\ -1, & t \in 2^{d-l} \cdot (2i-1, 2i] \\ 0, & \text{otherwise} \end{cases} \tag{4.4}$$

An example is shown in Figure 4.1. Keep in mind that this transformation is unambiguous and lossless.

Now we want to discuss what data is actually transformed. It turns out that using the unsmoothed, but logarithmic data works the best. The reason for this are the following: the $\log(x+1)$ transformation avoids the handling of too large values and as shown before results in a better value distribution. Now smoothing would certainly simply later compression attempts of the wavelet data, but it already prunes information. That alone is not a huge deal, but it gets problematic when we modify the wavelet data. Altering the coefficients of the wavelet tree, e.g., by rounding, data conversion, tree pruning (see section 4.2) or tree merging (see section 4.3); results in an unsmooth time series after restoring them from the wavelet data. So to use them afterwards a smoothing step is required. A pre-transform smoothing now would result in three steps of information loss while only smoothing after the information recovery makes parameters easier to tune and information loss easier to calculate.

4.2. Tree Pruning

The first attempt to reduce the amount of data is an obvious but simple approach: pruning of sub-trees that do not hold important data. This does not entirely reflect the approach of information loss we will use for the final algorithm, but it is a good milestone to grasp the idea. The idea was also presented in [49]. A tree node is pruned if the following conditions are met:

1. both children are pruned
2. setting its coefficient to 0 would not increase the error above a certain threshold

The error threshold is an application-specific parameter, similar to what video and audio compression codecs expose. Similar to these, the direct effect on the results is not always easy to guess since the decompressed data might be used for different purposes afterwards. For the method which extends the pruning to a merging as described in the following section, we measured the threshold-error relation in the context of neighborhood queries in section 6.2.

So pruning just equals the zeroing of the coefficient, but it allows us to remove the node entirely and save storage space since it does not contain any additional information than both pruned children and the coefficient that does not have an influence anymore. The question is in which order nodes should be tried to be pruned since this algorithm is greedy method and will stop immediately after the threshold is reached. We propose to shuffle the nodes first to avoid a bias to the beginning or end of the time series and then to sort the shuffled nodes starting the smallest coefficient. This makes it more likely to prune

Algorithm 1: pruneNode

Data: Node N
Data: Error threshold ϵ
Result: Pruned Node N, resulting error increase

```
1  begin
       /* Check if children are pruned                                  */
2      p_l ← isPruned(getChild(N, 0));
3      p_r ← isPruned(getChild(N, 1));
4      p_c ← p_l ∧ p_r;
       /* Calculate maximum error increase                              */
5      i ← 2^(d−l+1);
6      e ← |getCoefficient(N)| · √i;
       /* Prune if requirements are met                                 */
7      if e < ε ∧ p_c then
8          setPruned(N);
9          setCoefficient(N, 0);
10         return e;
11     else
12         return 0;
13     end
14 end
```

Algorithm 2: pruneTree

Data: Tree T
Data: Error threshold ϵ
Result: Pruned Tree T

```
1  begin
2      for l = d to 1 do
3          N ← getLayer(T, l);
4          shuffle(N);
5          sort(N);                                          // optional
6          for n ∈ N do
7              ε ← ε − pruneNode(n);
               // optional break if ε ≤ 0
8          end
9      end
10 end
```

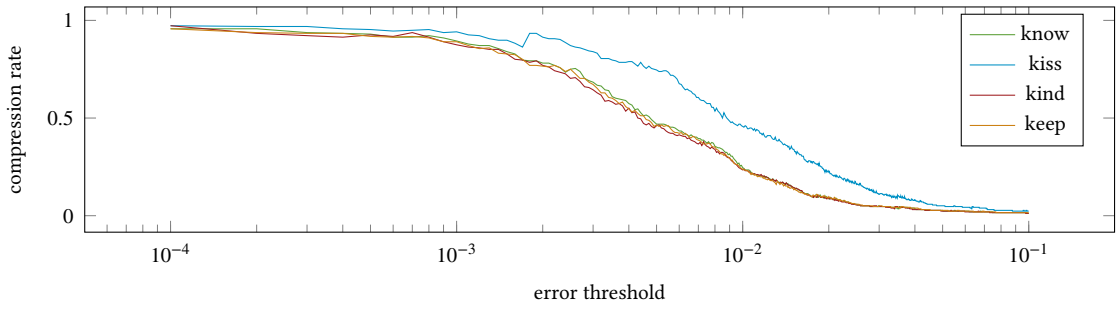

Figure 4.2.: Compression of example *n*-grams, without sorting/prioritizing

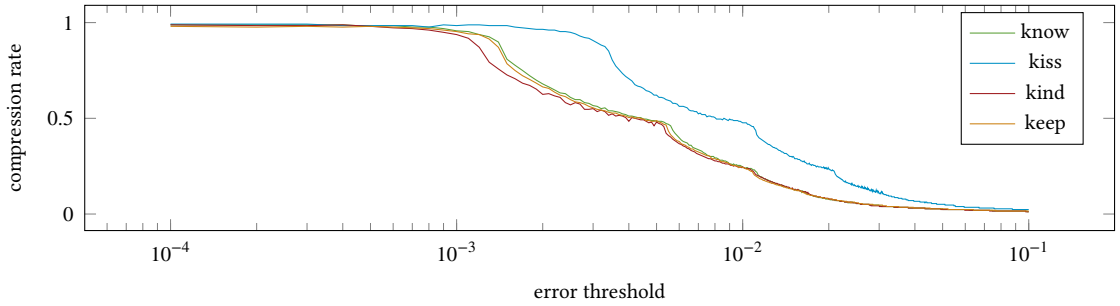

Figure 4.3.: Compression of example *n*-grams, with sorting/prioritizing

more nodes instead of just some with huge coefficients. Also, the results are more stable as you can see while comparing Figure 4.2 and Figure 4.3. There we show how the increasing error threshold affects the compression rate or in other words: the relative amount of pruned nodes as a function of the error threshold. The algorithm is shown in algorithm 2.

4.3. Tree Merging

An important thing to point out about tree pruning is that it does not exploit the fact that there are many time series that might have the same structure. The compression is purely executed on a single series. Tree merging uses a similar idea to tree pruning but with an important difference: instead of setting the coefficient to zero it will be set to a similar entry that is already present in the database (to which we come in a moment). Therefore, the subtree is not pruned but merged with another one. This can simply be done by altering the child pointer of a tree node. The error is calculated based on the difference of the time series resulting from the new tree and the original time series. A discussion of the error metric can be found in section 4.4.

Keep in mind to only merge nodes into others when their children are identical. Also, we want to point out that we do not merge superroots. This would in theory be possible but we rarely see the case that two time series are similar enough to make that possible. That complicates the implementation since we sometimes use the superroot to store additional information about the time series.

Algorithm 3: createMergeTask

```
Data: Node N
begin
    m_l ← isMerged(getChild(N, 0));
    m_r ← isMerged(getChild(N, 1));
    if m_l ∧ m_r then
        i ← getIndex(getChild(N, 0), getChild(N, 1));
        C ← findNearestNeighbor(i, N);
        if C then
            return newMergeTask(N, C);
        end
    end
    return 0;
end
```

Algorithm 4: addTreeToIndex

```
Data: Tree T
Data: Error threshold ϵ
begin
    q ← ∅;
    for l ∈ shuffle(getLeafes(T)) do
        push(q, createMergeTask(l));
    end
    E ← recalcError(T);
    while ¬isEmpty(q) do
        t ← removeTop(q);
        if isInErrorRange(E, ϵ, t) then
            update(E, t);
            execute(t);
            s ← createMergeTask(getParentOfNode(t));
            if s then
                push(q, s);
            end
        end
    end
end
```

The construction of the database of known trees[1] is solved as followed: the time series are added to the database in random order. One whole input tree of an entire time series is (partly) merged into the database, then the next one and so on. For the tree merging itself we changed from a strict bottom-up approach to a queue-based approach since that results in higher compression rates. The problem with the bottom-up approach is the following: it might be that there exist very similar subtrees that are cheap to merge but a strict level-ordered merging would result in prioritizing nodes next to the leaves of the tree. That is contradictory to our goal of merging as many subtrees as possible.

Still, the queue-based approach needs to start at the bottom and tends to be bottom-up. This is useful in a sense that the similarity we have defined is based on the gradient. Gradients are more influenced by the lower layers of the wavelet tree and so we merge this information first. This will be helpful for using the merged trees as an index structure.

The final algorithm is shown in algorithm 4 and algorithm 3. Notice that the creation of queue tasks does not involve the error threshold since we want to avoid expensive checks of the error bounds and only execute the corresponding routine when candidates are drawn from the queue. How this can be used to speed-up the error calculation is shown in the next section. An exemplary visual representation of how the merged results look like is shown in Figure A.1 and Figure A.2. Pay attention to the differences, which are the results of the different nearest neighbor query ranges.

4.4. Error Metric

The choice of the error metric, while being easy in many cases, opposes some questions in our case:

- How fast can a possible implementation be, not only in terms of complexity but also in terms of constant factors due to vectorization and cache layout?
- How good can it justify the quality of the decompressed data recovered from wavelet tree?
- How well does the decompressed data works with the later smoothing, gradient calculation and DTW?

We decided to try three different metrics during the process, which we will describe in the following subsections.

4.4.1. Summerized linear distance

The first metric, a linear absolute distance, is one that naturally occurs when looking for something easy to implement. With t and t' being the time series to compare, which both have a length of m, the summerized linear distance can be written as:

[1] To be precise: since the trees share common subtrees, they are not real trees anymore. We will continue to refer to them as trees since it is a simple and easy to understand term.

$$e_1\left(t,t'\right) = \frac{1}{m}\sum_{i=1}^{m}\left|t_i - t'_i\right| \tag{4.5}$$

The upper bounds can directly be calculated during the tree merging process since the influence regarding location and amplitude of certain tree nodes are known a priory. So the upper limit is simply the sum of the re-adjusted differences of the original node and the merging partner:

$$\bar{e}_1\left(t,t'\right) = \frac{1}{m}\sum_{m\in\text{merges}(t,t')} 2^{\frac{d-m_l}{2}}\left|m_x - m_y\right| \tag{4.6}$$

Here m stands for a merge task, which merges two nodes with the coefficients m_x and m_y, both located at level m_l. d stands for the depth of the corresponding wavelet trees.[2] All these terms are equal to the descriptions in section 4.1. Notice that all merges are included in the sum. So even when a node gets merged after both children got merged, they children are included as well.

While the implementation is rather fast because guessing of the upper limit does not require any knowledge of the actual time series difference, it is not well suited for all applications. A compression using this metric can lead to many jumps within the decompressed time series. Also, the metric tends to sacrifice huge local errors for small improvements of the overall compression rate.

4.4.2. Summerized quadratic distance

A way to put higher penalties on to huge local errors is a quadratic error function:

$$e_2\left(t,t'\right) = \frac{1}{m}\sum_{i=1}^{m}\left(t_i - t'_i\right)^2 \tag{4.7}$$

A drawback of this function is that the calculation requires the actual distance of the original time series and the compressed one:

$$\bar{e}_2\left(t,t'\right) = \frac{1}{m}\sum_{i=1}^{m}\bar{\delta}_i(t,t')^2 \tag{4.8}$$

Here $\bar{\delta}_i(t,t')$ is the upper limit of the distance between t and t' at the point in time i. This one can be guessed using a similar method like Equation 4.6.

$$\bar{\delta}_i\left(t,t'\right) = \sum_{m\in\text{merges}(t,t')} \mathbb{1}_{m\in i}\cdot 2^{-\frac{d-m_l}{2}}\left|m_x - m_y\right| \tag{4.9}$$

where $\mathbb{1}_{m\in i}$ describes the fact that the merging m does affect the point in time i. Notice the minus in the exponent. Since that upper limit strictly increases with at least the amount of the real delta during each merge, the error metric will reach the user-provided limit faster than expected. So we use a hybrid approach during the merging procedure: do cheap guessing until we reach the error limit and then calculate the real error and use this

[2] Notice that we mean the complete tree here, not only the subtree affected by the merge.

as a base for the following sum-up operations. The number of recalculations increases when slowly approaching the error limit but it is still way faster than calculating the real error on every merge.

4.4.3. Delta Range

Our last metric is an application specific one, which is based on the following observation: Shifting the entire time series up or down (in terms of the y-Axis) does not affect the gradient. Therefore, this should be allowed during the compression procedure. So we use the distance between the maximum of the delta and the minimum of the delta as an error function. Keep in mind that the delta itself is signed.

$$e_3\left(t, t'\right) = \max_{i=1}^{m} \left|t_i - t'_i\right| - \min_{i=1}^{m} \left|t_i - t'_i\right| \tag{4.10}$$

This metric has the same drawback as the one presented in subsection 4.4.2 but can guessed using a similar strategy:

$$\bar{e}_3\left(t, t'\right) = \max_{i=1}^{m} \left|\bar{\delta}_i(t, t')\right| - \min_{i=1}^{m} \left|\bar{\delta}_i(t, t')\right| \tag{4.11}$$

4.5. Tree as Index

Algorithm 5: traceDown

```
   Data: Nodes N
1  begin
2      D ← ∅;
3      U ← ∅;
4      for n ∈ N do
5          insert(U, n, 1, (a, b) ⟶ ab);          // start with weight of 1
6          if filterDown(n) then
7              append(D, getChild(n, 0));
8              append(D, getChild(n, 1));
9          end
10     end
11     if ¬isEmpty(U) then
12         traceUp(U);
13     end
14     if ¬isEmpty(D) then
15         traceDown(D);
16     end
17 end
```

Algorithm 6: traceUp

```
   Data: Nodes N
1  begin
2      U ← ∅;
3      for (n, w) ∈ N do
4          if filterUp(n, w) then
5              P ← findParents(n);
6              for p ∈ P do
7                  v ← w/|P|;                                  // split weight
8                  insert(U, p, v, (a, b) ⟶ ab);    // accumulate weight per parent
9              end
10             S ← findSuperroots(n);
11             for s ∈ S do
12                 handleSuperroot(s, w);
13             end
14         end
15     end
16     if ¬isEmpty(U) then
17         traceUp(U);
18     end
19 end
```

Besides providing compression, our tree construction also expresses similarity of time series in a way that it can be used as an index. To do so, we provide a generic tracing approach. Starting with the root of the query, we walk down the tree level by level to its leafs. At every level, we start trace back the nodes of the current level to all possible superroots. While the children during the trace down are unambiguous, the parents during the trace up approach are not, since subtrees can be shared by multiple trees and therefore can have multiple superroots. The two parts of the algorithm are also shown in algorithm 5 and algorithm 6. Besides emitting the superroots using a callback, we also provide filtering of the trace paths and some kind of weight accounting. Using this accounting calculates the chance ending up at a certain superroot if multiple random walkers would start at all possible nodes at the start of the trace up phase. It can be used to limit the number of possible paths. The filters can also be used to limit the tracer to specific parts of the time series or to stop the trace down earlier. This can be a huge advantage since higher error threshold during compression lead to too many possible superroots of nodes in the levels near the leaf nodes.

Our experiments will implement the following filter scheme:

- `filterDown`: drop branch if it leads to a time domain that was not requested for DTW; stop if a maximum depth is reached
- `filterUp`: drop branch if weight is below or equal a threshold

We leave an open call to find and apply more sophisticated filtering techniques.

4.6. Optimizations

There are three technical details that help to improve the compression ratio even further. The first one is the reduction of precision used to the coefficients. Instead of using full 64 bit floating point numbers we use 16 bit floats, also referred as “halfs” as defined by [50]. This increases the error due to conversion and leads to a lower amount of merges, but increases the overall compression rate in terms of memory size. The smaller size of the memory representation compensates the higher error. Using 16 bit numbers turned out to be better than 32 bit. There might be the possibility to use even less memory by using bit packing and changing the representation to a non-IEEE-compliant data type.

The second improvement is made by changing the representation of the child pointers within the tree nodes. Because we store all trees within a single memory-mapped region and this region may appear at different offsets during different program executions, we use offset pointers instead of real pointers. These offset pointers store the offset from the pointer itself to its target. Because the target is always located within the same mapped region, the offset stays constant no matter what global position the region has. Originally we stored 64 bit signed integer values as offsets. We reduced this to 48 bit values, which is enough to store $2^{48-1}\,\mathrm{B} = 128\,\mathrm{TB}$.

We want to point out that both changes of data types lead to a packed node representation of 112 bit which is acceptable on architectures that implement unaligned load and store operations. This is the case for x64. The packed layout only applies while the data is

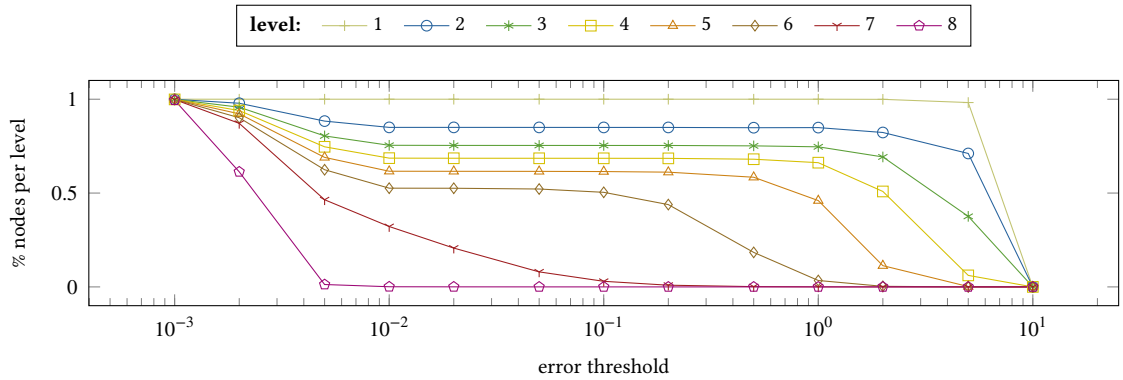

Figure 4.4.: Compression per tree level

located on disc or loaded into the RAM and cache lines. As soon as the values are fetched into processor registers, they usually require architecture-defined space. On x64 this are multiples of 64 bit.

The last one is a change on how the time series are shuffled before added to the index. Instead of doing a complete random shuffle, we separate the set of time series into chunks and shuffle these while preserving the order within the chunks. This improves memory locality and therefore results in better IO performance. The results vary from case to case and can be slightly worse than a full shuffle. We set the chunk size to 16, which we found a good trade-off.

A possible improvement closely related to reducing the size of the offset pointers is the removal of the null pointers stored in all leaf nodes. We did not implement this simply because it makes the code more complicated, also because there are not enough leaf nodes left after compression to justify this change.

4.7. Weaknesses

The main problem with our approach is its greedy nature. As shown in Figure 4.4, the merges usually do not reach very high levels. This is caused by the decreasing probability of merging with a node that belongs to the same large subtree. The further the merging process continues, the more likely it gets that somewhere in between there was a better node from a different subtree than nearest neighbor one. On the other hand we can not check all possible merges in all layers because the runtime complexity would grow exponentially with the number of indexed time series. So we limit the algorithm to a subset of decisions.

Figure 4.5 adds another detail to that problem. It shows how the compression rate develops when more data is added to the index. When using low error thresholds, the following can be observed: at the beginning, not enough data is present in the database so the first entries are not compressed very well. As more data is added, the compression rate improves. This improvement is limited by the fact that it gets harder to find the right merges that allow to merge larger subtrees. When using higher error thresholds this gets more obvious. Because the local merging is not limited by the threshold anymore, the

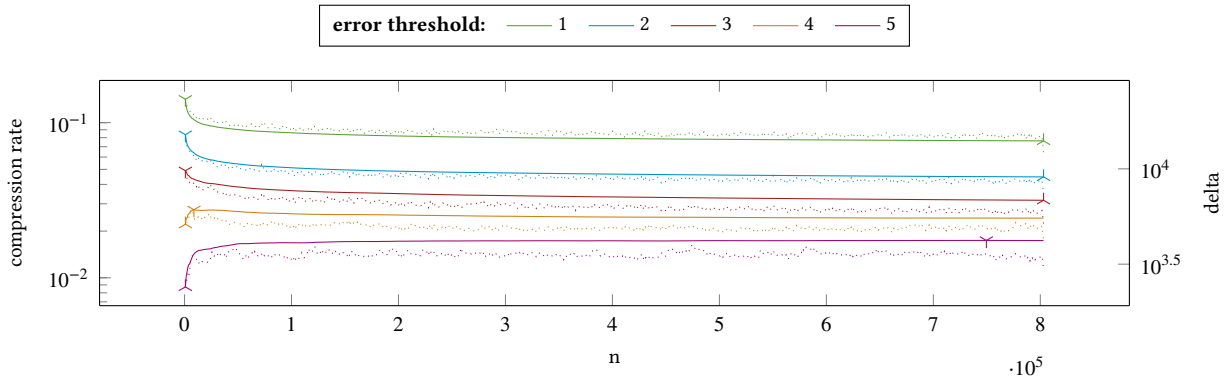

Figure 4.5.: Compression over the merging process for different error thresholds, with delta of #nodes as dotted representation, maximums/minimums are marked

choice of the right neighbor that allows to merge its parent as well plays a bigger role now. As a result the behavior flips and the results get worse the more data is added to the index, at least temporarily. When using such high thresholds it would be better to compress the data in chunks instead of the whole data set as one piece.

While the following section explain some of the fixes we have tried, we came to the conclusion that it might require a totally different strategy to find larger merges.

4.8. Failed Improvements

We tried different approaches to solve to increase the likelihood to merge with larger subtrees but did not succeed. This section explain the methods, the idea behind them and the results.

4.8.1. Subtree Index

Definition 5 (s-subtree)**.** *An s-subtree is a subtree starting at a certain node and that includes all child nodes and their subtrees. The size s equals the number of nodes in that subtree. Since in our case there are always two children*[3] *and the length of the time series is* 256*, possible values for s are:* $2^i - 1, 1 \leq i \leq 8$.

To increase the chance that the greedy algorithm chooses merges that belong to the same subtree we tried the following method:

1. select multiple nearest neighbors as candidates for a merge
2. do this for a node that belongs to the same 3-subtree (or larger ones) as well
3. check which pairs of merges belong to the same subtree and prioritize them

[3]Except for the superroot, which we just ignore here.

This requires that we are quickly able to figure out which nodes have which parents, so we introduced a subtree and superroot index to do so. Also we need to query multiple neighbors instead of just one. The index structure is the same as used for the tracer, but since compression requires way more index requests than the normal index usage, that approach is to slow for that application. Altogether that renders the entire approach impractical and we abandoned it. Also the compression results only improved slightly and sometimes even got worse.

4.8.2. FLANN

The main goal of our algorithm is to find the best matching subtree in terms of distance. To help the greedy algorithm to archive that task we tried to use FLANN ([51]) to merge subtrees of size 3 instead of single nodes. To do so we need to ensure that the coefficients stored in the tree nodes are pre-multiplied like the elements in Equation 4.6. It turns out that the huge amount of indices required, renders this method impossible to implement since it is not as memory and time efficient as plain sorted arrays. The result is that the calculation never finishes. Surprisingly the partial compression results are not even better, which might be due to the fact that merging entire subtrees tends to be less aggressive compared to handling single nodes.

4.8.3. Random Boosting

This improvement is similar to the one presented in subsection 4.8.1. We try to find better merges by observing more neighbors. This time we tried to gather a fixed amount of neighbors and select a random one with decreasing probability bound to the rank. Then we execute a complete subtree merge. This process is repeated multiple times and the best merge is kept. We also add a merge to the set where only the nearest neighbor is used to ensure that the greedy decision never gets worse due to the random elements, at least as a local outcome for that specific time series. We ended up with a heavily increased calculation time and less than 5 % improvements of the compression rate.

4.8.4. DTW

One of the main limitations of the current neighborhood search is that we are only looking for nodes that correspond to the same time span. This contradicts the later usage of the data through Dynamic Time Warping. So we tried to decouple the tree structure and subtree merging from the actual time dimensions. We limited the search for the child nodes to a time-monotonic warping and reused the implementation of the existing DTW. For the non-leaf-nodes we did not apply any constraint since the same children policy already forces merges with the correct level and correct warping. The results of this technique were not convincing since the additional degree of freedom during the merging makes it even more unlikely to hit neighbors that belong to the same subtree. So the compression rates got worse and we dropped this approach.

4.8.5. Timeless Index

One idea that may arise during investigating the results of DTW is that the time aspect of the index structure might not be that important at all. We gave this a try and removed the constraint from the leaf nodes entirely so that there exist a one-dimensional index for them. The inner nodes and the root node are only constrained by the same-children-policy. The results are, without surprise, pretty bad. The reason for this is that now the chance of finding large subtree-merges is even lower since merges on the leaf-layer are purely picked based on the coefficient and without any consideration of the consequences for the parent layer.

4.8.6. Pruning

Noise within the time series data results in small changes of the coefficients store in the tree nodes. Since our algorithm only uses the nearest neighbor as a candidate for a merging operation, these small changes quickly result in different decisions during the local merging process. This is suboptimal since our method should rather focus on the big picture rather being to sensitive to noisy input. We tried to solve this problem by introducing additional information pruning, which we hoped should eliminate the noise sensitive, in three variants:

1. before merging
2. during merging
3. after merging

The pruning zeroes low significant parts of the floating point coefficients, bit by bit, so with increasing error. So the ratio of the information loss stays approximately constant no matter what size the coefficient has. The idea behind this is to engage the algorithm to ignore noise within the coefficient and to have fewer decisions to make. On the other hand this pruning also increases the error without resulting in any compression at all and is only useful for entries that will be added to the index afterwards. Sadly the increase of the error rate overweights the desired affect of helping the greedy algorithm and therefore this approach does not have a positive effect, in all three variants.

Keep in mind that this type of pruning does not decrease the memory requirements of the tree nodes since there is no fixed cutoff but rather a individual decision how much information should be kept. So to than decrease the node size by storing a shorter representation of the coefficient you would also need to store how much information got pruned, which takes so much space that it eliminates its purpose. For technical reason it is also quite difficult to store nodes with sizes not fitting into whole bytes.

4.8.7. Seeding

As shown in Figure 4.5, normally the first elements inserted into the database are not compressed very well. This is due to the fact that there do not exist enough possible

subtrees to merge with. An idea to improve the initial compression rate is to insert some seeds into the database before starting compression. There are multiple possible types of seeds:

1. randomly chosen or by metric selected elements of the dataset
2. fixed value entries, e.g. zero
3. time series generated by sinus/cosinus waves or combinations of them
4. fractals

While choosing elements of the dataset is considered cheating because we would add data to the database without counting it, the other methods would in general work because the description of the corresponding trees could be considered as part of the model and are fixed for all compressed data sets. The main issue is that someone needs to come up with a proper choice of this seeds and we were unable to figure out a proper way to do this.

4.9. Different Ideas

After investigating failed approaches we want to present some ideas that we did not implement but that we think are worth to consider.

The first alternative is to instead compressing the actual time series data, to compress lower and upper limits within the warping as presented in [15]. The question then is how to design a tracer that handles both trees at the same time. Another open point is if both limits should be added to the same index or not. Using the same index would probably enable higher compression rates while it requires the tracer to handle the mixed data set.

Another idea is the usage of another wavelet type to do the decomposition, e.g. from the set of Daubechies wavelets described in [52]. This would lower the depth of tree and possible allow larger merges. On the other hand, this also leads to more children per node, which lowers the chance that a specific node can actually be merged. It also increases the amount of pointer information stored within the tree and therefore lowers the compression rate, at least when implemented in a naïve way. It also increases the complexity of the implementation. With all this factors, it must be carefully considered if this approach is worth a try.

The next idea is to use a general-purpose lossless compression algorithm to compress the data. To do so, the data should be split in chunks so it is still possible to provide quick random access. We did a quick experiment with 1 MB chunks so one contains 512 time series. We then test two possible ways to compress the data. The first one is to compress the floating-point data that is the input for the DTW. Because we know that floating-point material is usually compressed badly, we also try to compress the initial data set, which contains unsigned integer data. In that case, the $\log(x + 1)$ transformation, the smoothing and the gradient calculation must be evaluated on-the-fly. The results for different compression algorithms are shown in Table 4.1. They usually have very different performance characteristics in terms of memory and time requirements, but we did not

algorithm	compression level	average compression rate	
		floating-point data	integer data
snappy[a]	—	70.41 %	18.93 %
lz4[b]	1	68.75 %	21.15 %
	5	66.87 %	13.63 %
	9	66.86 %	12.27 %
gzip[c]	1	65.47 %	12.17 %
	5	64.55 %	11.00 %
	9	64.45 %	10.06 %
xz[d]	1	60.17 %	7.21 %
	5	59.93 %	7.07 %
	9	59.89 %	6.81 %
brotli[e]	1	62.98 %	11.35 %
	5	61.61 %	9.36 %
	9	61.24 %	9.31 %

Table 4.1.: Lossless, general-purpose compression algorithms

[a] snzip (`https://github.com/kubo/snzip`) version `1.0.3` with `framing2` compression format
[b] lz4 (`http://www.lz4.org/`) version `128`
[c] gzip version `1.8`, uses the format described in RFC 1952
[d] xz (`http://tukaani.org/xz/`) version `5.2.2` with liblzma version `5.2.2`
[e] brotli (`https://github.com/google/brotli`) version `0.3.0`

include these factors into the evaluation. Our experience showed that, even when the integer data enables higher compression rates, it does not imply that it is faster or slower to compress and decompress. The compression rate might be improved by sharing information between the chunks while still allowing decompressing them separately. A factor that is important when choosing a compression algorithm is that in this application, the data is compressed once and decompressed many times, so fast decompression is required while resources required during compression are a less important factor. This is also the reason why we have included brotli into the comparison. This method, mainly pushed by Google, is usually used for asset compression on web servers and fast decompression on web clients. It is also to decide how decompressed data is cached and when the cache, if implemented, should be cleared. Technology borrowed from projects like zram[4] and zswap[5], both implemented within the Linux kernel, might be useful.

It might also be a good idea to evaluate the usage of other data types for storing the time series data. For integer data this could mean to dynamically select the bit-width of the type according to the largest value within the whole data set, withing a chunk or withing a single time series. For floating-point data this means to either use a smaller IEEE type, similar to the design decision we have made in section 4.6 or to use a special type that better exploits our data ranges. For example we think that 5 bit for the exponent are too much. Completely different data types like the A-law and μ-law as defined in [53] might

[4] `https://www.kernel.org/doc/Documentation/blockdev/zram.txt`
[5] `https://www.kernel.org/doc/Documentation/vm/zswap.txt`

be worth to try. Stripping the sign from the floating-point data could also be possible since the resulting curves should still preferable match against the same neighbors.

5. Implementation

An important aspect of our work is the development of a performant and reusable implementation. In this chapter we are going to explain this work, generic strategies as well as specific solutions that we have applied to achieve that objective.

5.1. A KISS approach

After considering multiple ways of implementing a high-performance algorithm, we decided to use C++14 as standardized in [54] and under partial application of the recommendations of [55] and [56]. This gives us the ability to have fine control over memory allocations and data structures as well as the opportunity to write modern, well-structured and reusable code. In combination with a modern compiler this results in native executables that exploit a wide variety of instruction sets of current CPUs.

Instead of developing the algorithms as a single monolithic block we designed them as reusable executables with single scopes similar to the system tools provided by UNIX-like systems. An overview of these tools and their interaction can be found in Figure 5.1. There purposes are:

- create: creates a new matrix-like storage file with given size and filled with 0 values
- dump_index_wavelet: decompresses wavelet tree data and stores resulting time series data back into an uncompressed matrix
- extract_stems: extracts list of normalized words from a word-to-stem transformation file
- filter: filters list of *n*-grams according to the pattern described in subsection 3.1.2
- iJulia: a generic tool to analyze and visualize data and to prototype algorithms
- index_dtw: indexed data as described in section 3.3
- index_wavelet: transforms time series data into trees, merges them and stores the result together with a child-to-parent index into a file
- normalize: normalizes strings as described in subsection 3.1.3
- print_wavelet_tree{,2}: extracts multiple wavelet trees and their merges from a wavelet index and outputs a DOT or TikZ file; the latter one is used to produce the trees shown in the appendix

Figure 5.1.: Used tools

- prune_support: prunes list of possible normalized *n*-grams according to subsection 3.1.5

- query_dtw_indexed: queries the nearest neighbors, accelerated with an indexed produced by index_dtw and described in section 3.3

- query_dtw_simple: queries the nearest neighbors with a simple bruteforce approach

- query_dtw_wavelet: queries the nearest neighbors, accelerated by an indexed produced by index_wavelet

- scan: reads input data from Google and stores all *n*-grams as well as the range of years and the value ranges present in the given file

- stem: applies word normalization as described in subsection 3.1.4 and stores the transformation into a file; this is done because this step is usually to slow to do ad-hoc in contrast to the string normalization

- store: stores data from Google input files into a matrix created with create; applies string normalization and word normalization from a transformation file and correctly adds overlapping results

- transform: generic transformation of time series data, e.g., to execute $\Delta_1(\log(x+1))$

Parameters are usually passed via command line arguments and data is stored either in text files or in case of data matrices as C-like arrays. Data matrices are stored in row-major order with every time series stored in a single row. This enables us to use memory-mapped IO so the operating system with its global and complete view of system resources can decide about memory management. As a result we get caching between program executions and good behavior in case of low-memory situations[1]. Another advantage of storing the data this way is that it can be loaded by other tools and programming languages, e.g., for visualization purposes. So we used Julia[2] to run quick analysis task and to produce plots and smaller reports[3].

Common code is shared by header files and static libraries and executables are linked statically. While this increases their size, it enables easy deployment of binaries compiled on a developer machine, which includes the compiler suite, to a server, which has fewer packages installed and might even be equipped with a different C library. To simplify development and reusability the compilation process is managed by CMake, which also ensures that most required libraries are downloaded and compiled on-the-fly. In theory this also enables cross-compilation for different architectures like ARM64.

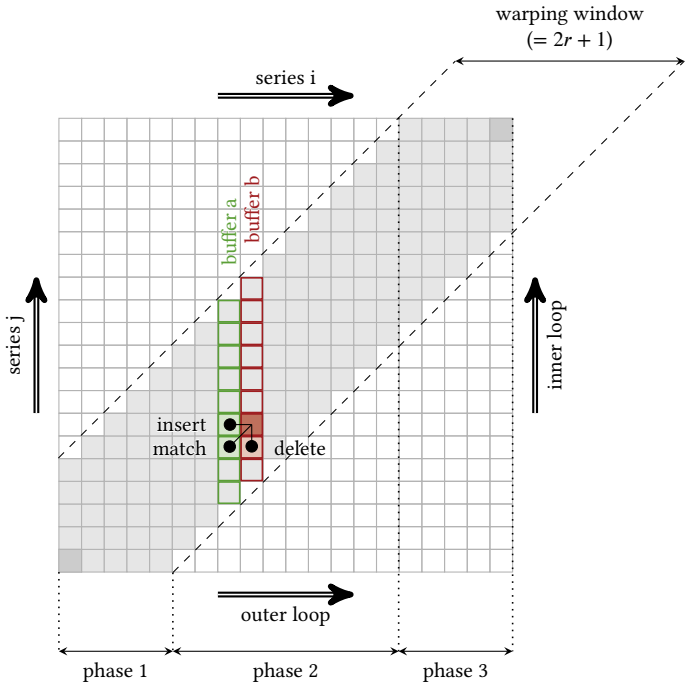

Figure 5.2.: Fast DTW implementation

5.2. DTW

One of the most important parts of our implementation is having a solid and fast DTW baseline. The fundamental idea behind this is explained in Figure 5.2. Because Dynamic Time Warping is an optimal dynamic programming method, we only need to store the current set of alternatives instead of the entire history of possible warping paths. Furthermore, our warping window is limited to a Sakoe-Chiba Band of size $2r + 1$. This leads to the possibility to use a double buffering technique — one buffer for the old set of optimal paths and one buffer for the new set. We also do not need to store the actual warping path but only the optimal distance, which reduced memory allocations and speeds up the implementation even further.

As shown withing the illustration, every loop iteration, as well of the outer as for the inner loop, depend on the result of the last iteration. So it is not possible to parallelize the loops. That means that the optimal warping path for a single time series pair has to be calculated linear with a single thread. Luckily we only use a single query time series and calculate the DTW against all other series. These calculations are independent and their execution path does not depend on the actual input values, which gives us two possible orthogonal optimization strategies. First we could load multiple time series at once and use vectorization to calculate the DTW for all of them. This requires a somewhat modern

[1] This should not happen during in an optimal setup but might occur due to bugs or during testing on development machines.

[2] `http://julialang.org/`

[3] We recommend a combination of Julia, iJulia and Gadfly for that.

processor. In our case we exploit AVX2[4] and FMA[5] instructions of current x64 processors. The first instruction set ensures that we can operate on vectors of 256 bit, which equals 8 32 bit floating point operands, in parallel on one core. The other one enables us to get results of combined multiply and add operations in less cycles. The second optimization strategy, which is straightforward is to use multiple cores to calculate independent DTW results. In the end we partially sort these results and emit the nearest k neighbors. Because we intend to use the implementation on a server system with multiple querying users, we did not implement the multi-core approach, but our generic code would allow us to do so very easily.

Another optimization is the reduction of the precision. Originally we intended to use 64 bit floating point numbers during the DTW calculation. We reduced that to 32 bit since our time series only have a length of 256 and therefore the results are precise enough for our users.

During the implementation we took special care of efficient memory management. During the entire DTW calculation no dynamic memory is allocated or freed. We create all buffer once and reuse them during the execution. This reduces the interaction with data to input, buffers, and output memory, which increases cache efficiency and overall performance. The code uses a strategy pattern to implement the control flow driver once and providing optimized plugins for single comparisons and vectorized inputs. It is possible to use the same driver to also store warping path or other metadata from the execution. Because the strategy is using templates instead of polymorphism, there is no runtime overhead.

An improvement we have tried but dumped was the early rejection of possible neighbors. Because we only use the k nearest neighbors it should be possible to maintain a heap with the best k candidates and stop the DTW calculation for new ones if all possible paths already exceed the distance to the farthest candidate. This works in theory but shows some problems during the real world tests. To do this early stop efficiently, the breaking conditions must be met by all time series that are handled in parallel by the vectorized engine. This delays the break in many cases and introduces additional checks. Also, the maintenance of the queue is slower than calculating all distances and doing a partial sort. In the end the improvement is slightly slower than the naïve full calculation of all distances.

5.3. Tree Merging

We now explain, which decisions where made during the implementation of the main algorithm. Some details are already listed in section 4.6 and we do not repeat them here.

The tree is implemented in a way that every node only stores its coefficient and two child pointers with the child pointers of the leaf nodes being null. A similar concept is implement for the superroot. Apart from the anchor value and the root pointer we also store the final error value of the compression and an integer value representing the n-gram the superroot belongs to. The child-to-parent and root-to-superroot indices are stored

[4] Advanced Vector Extensions
[5] Fused Multiply–Add

as hash maps, mapping node pointers to vectors. Since pointers change during program execution, all pointers are stored as offset pointers and for hash maps we calculate the offset to a global anchor before hashing, so that the hash is independent from the offset value stored in the pointer as well as from the global address.

During the index process additional data structures are needed. Indices to lookup the nearest neighbors are implemented as sorted vectors and binary searches. Additionally, coefficients are inlined into the index so the vectors contain coefficient-pointer pairs. This construction preserves memory locality and efficiency. For root nodes and inner nodes we store a hash map mapping two children to one index. This ensures that only neighbors that only have the same children than the merging candidate are stored in an index. For leaf nodes we use the time-relative position of the leaf (0 is the very left leaf and 255 the very right one) as keys to retrieve the index. The correct position of the leaf nodes and the correct child-parent relationship ensures that inner nodes also have the right position. Modifications to that index where made for possible but failed improvements, e.g., in subsection 4.8.4. Furthermore, we store counters for the number of stored nodes to output statistics and compression rate information.

There is no special serialization handling. Instead, we just use a memory mapped file and the Boost Interprocess library to allocate and manage memory withing that file. That means that the index can only be used on machines of the same endianness. We usually use oversized index files to ensure that we do not need to grow files during operation, which usually requires remapping it, and to speed up memory allocation. There might be more efficient storage implementations but since our algorithm has generic weaknesses, we did not try to improve this part further. As for the matrix data we also rely on the operating system to handle low memory situation and paging for our index data.

Another aspect of our algorithm is randomness. We use fixed seeds to ensure reproducibility and to simplify bug hunting. Keep in mind that the results might change depending on the stdlib implementation. Hash maps are not initialized with random seeds, which can affect security for real world applications. We encourage users of the code to re-evaluate the entire package in terms of security before usage.

5.4. Alternatives

An alternative, which we think is worth mentioning, is implementing the entire software stack in Rust[6] instead of C++. We would expect equal software performance and clearer code while reducing the number of possible bugs. The reason we did decide against it was, at the time of writing, the lack of proper advanced memory allocation as we use it for storing data structures in memory mapped files. This does not prevent us from recommending Rust as a tool for high performance data analysis, since we already have good experience while implementing other kind of algorithms. Rust could especially be a good choice for less-trained programmers since C++ often results in accidental memory corruptions when code is programmed by these kind of people. Rust with its novel type system and compile time checks could catch these bugs while still allowing unchecked operations by explicitly declaration.

[6] `https://www.rust-lang.org/`

6. Evaluation

We now want to evaluate the different parts that we worked out before. We start with a definition:

Definition 6 (Parameter/Filter response)**.** *A parameter or more specific a filter response describes how an adjustable variable influences a measurable value, e.g.:*

- *A filter influences the error made by an algorithm in a linear way, iff the error can be described as $f(x) = ax + b$ in terms of the filter value x and two constants a and b.*[1]
- *The maximum error of the compressor has a monotonic response to the compression rate. This means that if the maximum error is increased, the compression rate does not decrease.*[2] *This does not apply any additional attributes like that the compression rate is continuous in terms of the maximum error nor that it can be even expressed as a function of it.*

The response curve is the graphical representation of the measured value plotted as function of the adjustable parameter.

Next we want to check if our baseline selection makes sense by check the neighborhood search for semantic meaning. Afterwards we check how our compression algorithm affects the DTW calculation. We continue with testing different parameters of our tracing algorithm and finish with some obligatory Performance measurements.

6.1. Baseline Sanity

Before we start to evaluate the actual algorithms, we show that our baseline is sane. Table 6.1a shows the top 20 neighbors, in terms of the DTW distance[3], of the 1-gram "drug" with a warping radius of $r = 10$, measured over the full 256 years. The auto-cutoff-point as described in subsection 3.2.4 is marked. For these examples, only neighbors from the 1-gram data set are shown. Neighbors are represented by their normalized placeholder, which is derived as illustrated in section 3.1. Therefore, they are not necessarily real words and can represent multiple original entries within the data set. For example, "treatment" represents "treatment", "treatmental", "treatmente", and "treatments". The neighbors are

[1] This is not the case for our algorithm.

[2] That is also not the case for our algorithm because of its greedy nature. It is true in when the response curve is smoothed though.

[3] As a reminder: This is the normalized square-root of the sum of the squared distances when using the optimal warping path. The normalization is the devision by square root of the number of years within the requested time range. This ensures easier comparability.

rank	*n*-gram	distance
0	drug	0
1	treatment	0.033 464 6
2	impair	0.034 129 4
3	fundament	0.034 622 6
4	scar	0.034 844 4
5	tract	0.035 141 4
6	conjunct	0.035 203 3
7	univers	0.035 246 5
8	demonstr	0.035 316
9	western	0.035 318 7
10	li	0.035 393 1
11	mayb	0.035 446 3
12	sex	0.035 460 3
13	promot	0.035 463 6
14	member	0.035 473 9
15	blood	0.035 580 6
16	action	0.035 716 2
17	malign	0.035 750 6
18	counter	0.035 784 3
19	formal	0.035 895 4

(a) [1, 256]

rank	*n*-gram	distance
0	drug	0
1	character	0.018 666 9
2	abnorm	0.019 763 1
3	recurr	0.019 771 8
4	dissemin	0.020 063
5	retent	0.020 266 9
6	primari	0.020 359 2
7	transplant	0.020 496 5
8	pregnant	0.020 604 3
9	tension	0.020 758 5
10	southeast	0.020 773
11	matern	0.020 839 6
12	crucial	0.021 014 9
13	region	0.021 018
14	clinic	0.021 034 4
15	specif	0.021 054 5
16	proxim	0.021 084 9
17	relax	0.021 178 2
18	morpholog	0.021 181 1
19	vulner	0.021 248 2

(b) [129, 256]

Table 6.1.: Neighbors: 1-grams, $r = 10$, “drug”

words that are related to the concept “drug” with some exceptions, for example the neighbor “li”, to which no other word was transformed during the normalization and stemming procedure. The Google data set just contains “li” as a word and we are not aware of a proper interpretation of this word. The meaningfulness increases when limiting the DTW to the second half of the data set. This is shown in Table 6.1b. It also turns out that “drug” is used in two contexts: as medicine and in terms of narcotics.

The warping radius plays an important role when extracting good results from the data. Table 6.2a and Table 6.2b show too low and too high radius values and how in the first case the neighbors tend to be very pointless while in the latter case the results seem to be very common words.

Another interesting but logical observation is that common verbs like “know” result in unenlightening results as shown in Table 6.3a. On the other hand nouns like “war” show sane and meaningful data. This is shown in Table 6.3b.

The situation changes when we try to query phrases and add 2-grams to the data set. As Table 6.4a shows, results may contain more entries that do not provide any insights. The data in Table 6.4b are even worse. At least for us, no insights are extracted at all.[4] When limiting the search to the 1-gram database, the results are even worse. Some reasons for this are that our cleaned up data contains a lot of words that do not provide direct

[4]People might be able to interpret the “zhang daol” entry, which is derived from the name “Zhang Daoling”.

rank	*n*-gram	distance
0	drug	0
1	treatment	0.051 692 5
2	follow	0.052 971
3	new	0.053 491 9
4	s	0.053 782 1
5	enlarg	0.054 032 9
6	press	0.054 096 2
7	about	0.054 134 5
8	use	0.054 172 9
9	sever	0.054 192 3
10	flow	0.054 294
11	low	0.054 429 4
12	high	0.054 491 4
13	avoid	0.054 507 4
14	daili	0.054 518 4
15	for	0.054 555
16	skill	0.054 617 4
17	remov	0.054 691
18	chang	0.054 728 3
19	time	0.054 731 8

(a) $r = 0$

rank	*n*-gram	distance
0	drug	0
1	treatment	0.033 463 6
2	fundament	0.033 599 9
3	impair	0.033 934 1
4	member	0.033 942 7
5	demonstr	0.034 188 6
6	scar	0.034 695 1
7	region	0.034 724 2
8	tract	0.034 853 5
9	mayb	0.035 096 5
10	becaus	0.035 125 6
11	current	0.035 130 4
12	degener	0.035 140 6
13	malign	0.035 153
14	conjunct	0.035 174 3
15	western	0.035 214 5
16	univers	0.035 246 5
17	blood	0.035 328
18	educ	0.035 359 7
19	li	0.035 393 1

(b) $r = 20$

Table 6.2.: Neighbors: 1-grams, $[1, 256]$, “drug”

insights. For example the words “the” and “of”, which are listed in some 2-gram neighbors, do not provide meaningful data. On the other hand the same words may be important for 3-grams, e.g., in “Medal of Honor”. The reason that so many neighbors are found that seem to contain random data is the following: the more data is stored in the database, the more likely it is to find neighbors that have a lower distance. A similar effect can be observed when dealing with correlations of real world data. [57] contains a list of these cases that, with a very high probability, do not belong to the same cause. This is a general issue with methods that try to extract links between words or concepts based on usage count. We were aware of this problem beforehand and it is discussed in section 3.2. Possible workarounds can be:

- Increasing the pruning threshold: This should eliminate many pointless entries that only do not represent a common trend.
- Weighting distance with *n*-gram importance: It should be possible to weight the distance with additional information about possible neighbors, e.g., with their total counts or with linguistic metrics on how common or special words are.
- Extend filtering/normalization to take inter-word information into account: Currently prepositions and other non-important words in 2-grams and at the end of 3+-grams often do not provide value information.

rank	*n*-gram	distance
0	know	0
1	do	0.009 733 17
2	never	0.010 465
3	not	0.010 611 2
4	what	0.011 191 6
5	out	0.011 269 1
6	how	0.011 427 3
7	down	0.011 487 7
8	no	0.011 639 5
9	think	0.011 674 4
10	own	0.012 321 8
11	away	0.012 352 9
12	mistak	0.012 384
13	that	0.012 566 1
14	hear	0.012 628
15	as	0.012 651 1
16	neither	0.012 681 3
17	so	0.012 688 2
18	one	0.012 772
19	too	0.012 795 5

(a) “know”

rank	*n*-gram	distance
0	war	0
1	fight	0.021 591 8
2	peac	0.023 811 7
3	freedom	0.024 415
4	enemi	0.024 764 3
5	impend	0.024 997 5
6	steadi	0.025 059 5
7	militari	0.025 343 8
8	nation	0.025 674
9	peopl	0.026 238 2
10	countri	0.026 242 3
11	attack	0.026 319 7
12	arm	0.026 451 3
13	threaten	0.026 457 1
14	ralli	0.026 495 2
15	battl	0.026 500 6
16	suppli	0.026 852 5
17	violent	0.026 884 2
18	forese	0.026 989 6
19	over	0.027 005 9

(b) “war”

Table 6.3.: Neighbors: 1-grams, $r = 10$, $[1, 256]$

- Use multiple search queries and calculate a (weighted) distance to all of them: An example is shown in Table 6.5 where we add “philosophi” as a second query. There exist multiple methods on how to do that, in our example we just used the sum of the DTW distances, but Euclidean distances may also work as well as more sophisticated ways like running a DTW against multiple time series at once.

6.2. Compression Distortion

Because our compression method leads to information loss, we need to figure out how that loss affects the data in terms of different use cases. We have build multiple methods to compare the compressed data with the uncompressed one and will describe them in the following subsections.

6.2.1. User-facing data

First we want to figure out how compression affects the data that is perceived by the user. We use the output as described in subsection 3.2.4 and plug in the optional compression before the smoothing as it was discussed in section 4.1. We choose a single cutoff point for

rank	n-gram	distance
0	think therefore i am	0
1	diversif from	0.067 440 5
2	term leadership	0.074 937 3
3	temporarili avoid	0.075 077 7
4	fellow survivor	0.075 602 4
5	relev result	0.075 639 5
6	from subpoena	0.076 452 8
7	yet unsur	0.076 553 7
8	generat clear	0.076 751 8
9	arson charg	0.076 759 6
10	fieldwork have	0.076 832 2
11	cultur despit	0.077 026 1
12	fewer manag	0.077 072 3
13	poor weight	0.077 174 3
14	fund student	0.0772
15	stanley miller	0.077 463 3
16	requir technolog	0.077 533 5
17	particip keep	0.077 555 2
18	into gene	0.077 573 1
19	saxophon to	0.077 592 3

(a) “think therefore i am”

rank	n-gram	distance
0	logic takes care of itself	0
1	keratomileusi for	0.055 544 4
2	unicompartment knee	0.056 213 7
3	the fasttrack	0.057 103 3
4	of mcts	0.058 616 4
5	base lube	0.058 743
6	pareto rank	0.059 210 6
7	basket peg	0.059 318
8	to rauschenberg	0.059 652 9
9	carcinoid	0.060 003 5
10	jiceng	0.060 295 6
11	cpe cours	0.060 473 3
12	the tetrodotoxin	0.060 494 2
13	dibartola	0.060 643 2
14	qujiang	0.061 077 9
15	sequenti assembl	0.061 59
16	engag scholarship	0.062 214 4
17	merrienbo	0.062 344
18	zhang daol	0.062 384
19	combust environ	0.062 389 3

(b) “logic takes care of itself”

Table 6.4.: Neighbors: 1 + 2-grams, $r = 10$, $[129, 256]$

both, the uncompressed and compressed, results since the choice is very unstable under small compression artifacts. We then calculate the following two metrics:

1. We treat the sorted list of the nearest neighbors as strings and calculate the Levenshtein distance ([58]) of them.

2. We measure the amount of new entries produced by the compressed version compared to the uncompressed reference as a relative measure.

As shown in Figure 6.1 the set of the nearest neighbors and the ranking of these neighbors is very unstable as soon as we introduce some noise to the data. This makes our algorithm rather bad when the output data is directly presented to the user. This situation might be improve when combining the distance to multiple queries to one final score, which might be required for handling groups of words or entire concepts as a search query.

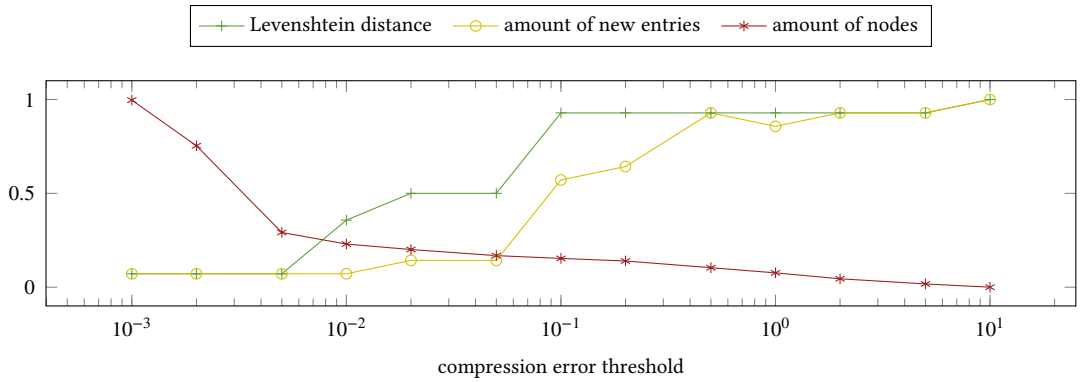

Figure 6.1.: Compression distortion, user facing data, query is “drug”

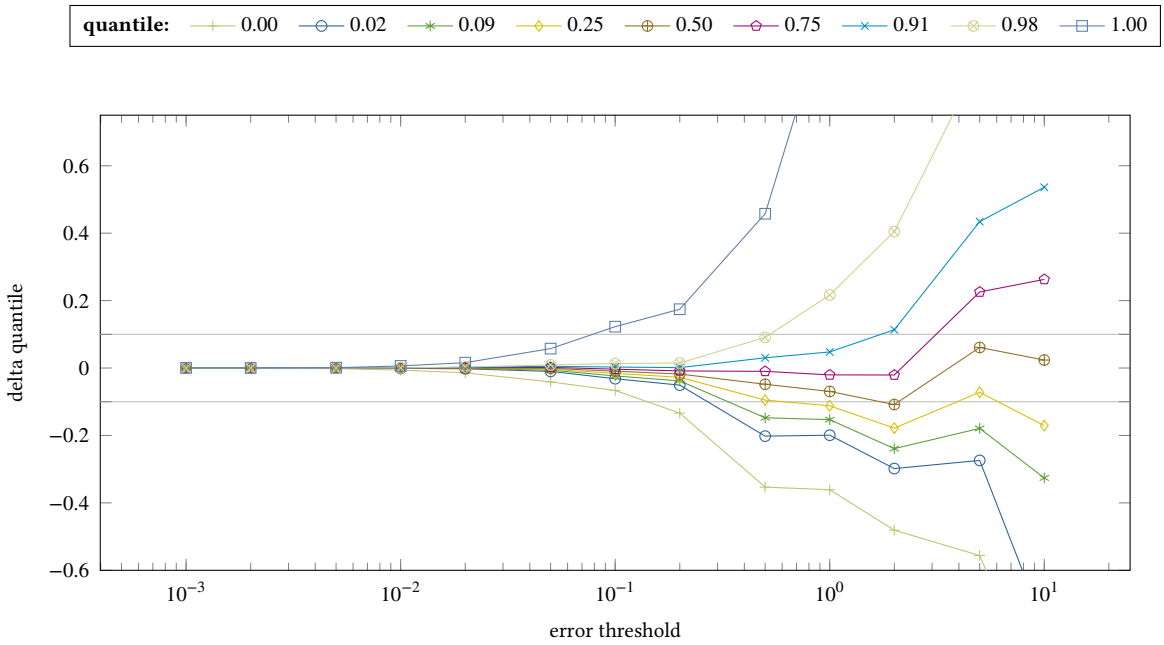

Figure 6.2.: Compression distortion, all distances, query is “drug”

rank	*n*-gram	distance
0	basic introduct	0.117 28
1	god beyond	0.117 417
2	underground chamber	0.117 425
3	all biblic	0.117 59
4	stori add	0.117 698
5	less put	0.118 025
6	would altern	0.118 063
7	also proof	0.118 091
8	for tighter	0.118 122
9	pay consult	0.118 165
10	simpli admit	0.118 373
11	child requir	0.118 39
12	aloud what	0.118 508
13	those think	0.118 584
14	univers scienc	0.118 605
15	all sixti	0.118 674
16	isaac in	0.118 777
17	selfconsci of	0.118 791
18	also imit	0.118 859
19	some consult	0.118 875

(a) “think therefore i am”

rank	*n*-gram	distance
0	via hematogen	0.099 445
1	messag share	0.1004
2	creatur	0.100 961
3	fof	0.101 037
4	hree	0.101 637
5	consult typic	0.101 826
6	hypoxia to	0.101 862
7	irom	0.101 975
8	of admiss	0.102 331
9	vate	0.102 352
10	nli	0.102 393
11	reflect dietari	0.102 469
12	hap	0.102 618
13	economi surg	0.102 829
14	fold plastic	0.103 087
15	therapist obtain	0.103 215
16	possibl point	0.103 358
17	unlimit amount	0.103 394
18	cide	0.103 399
19	paranorm in	0.103 44

(b) “logic takes care of itself”

Table 6.5.: Neighbors: $1+2$-grams, $r = 10$, $[129, 256]$, linearly combined with “philosophi”

6.2.2. All distances

To measure how the compression affects the actual distance, we calculate the distance to a fixed query and calculate the normalized distance of the original result and the ones produced by the compressed data:

$$\delta_i(a, b) = \frac{b_i - a_i}{a_i} \tag{6.1}$$

Then we measure different quantiles of this distance as plotted in Figure 6.2. We also annotated the markers for $-10\,\%$ and $10\,\%$ to show when, in our opinion, the distortion gets unacceptable for different amounts of time series. It turns out that even with some outliers, the overall result is more stable than the user-facing data. Therefore we conclude that for some applications higher compression rates and the corresponding artifacts can be acceptable.

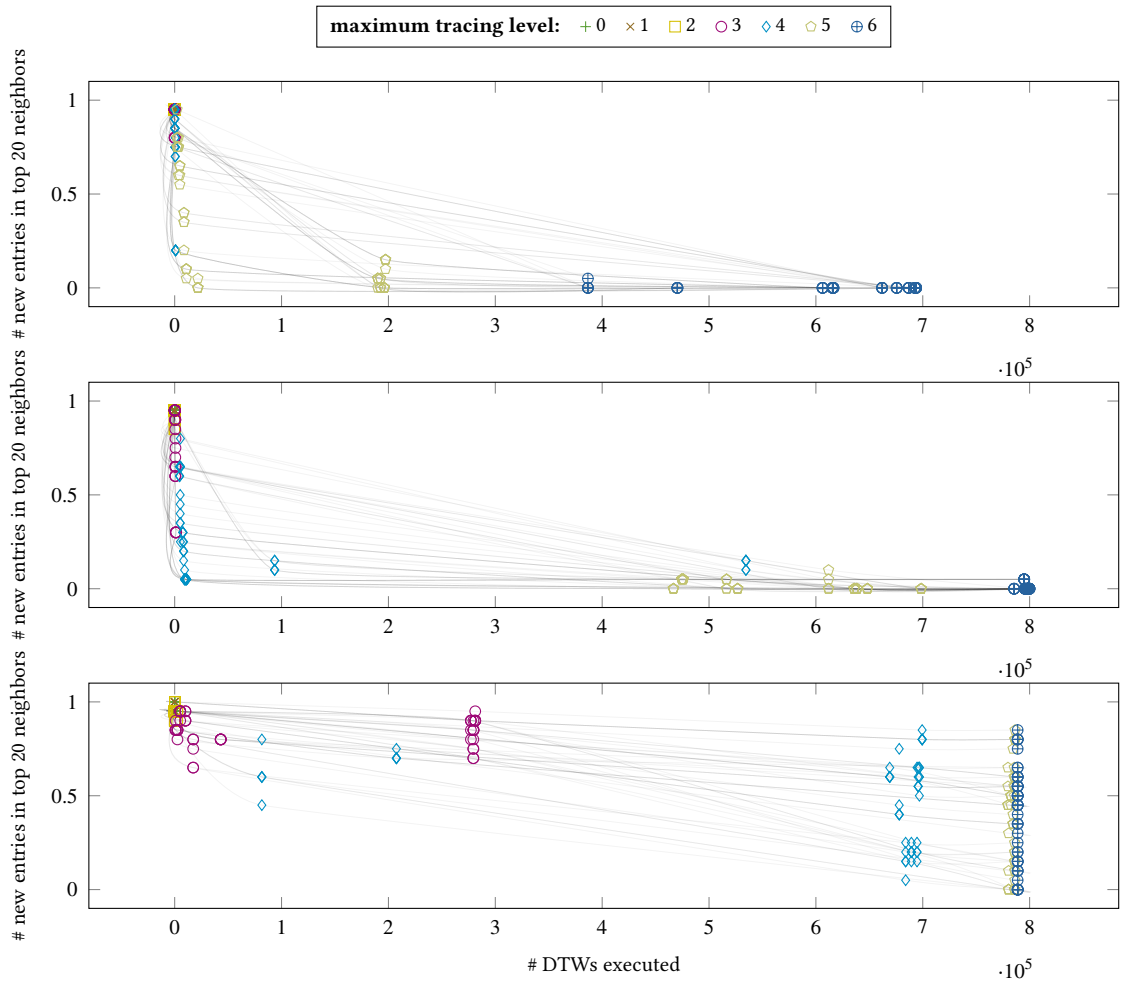

Figure 6.3.: Efficiency of traceback-based indexing, testing trace-down depth and maximum compression error

6.3. Quality of the Tree-Index

We now want to evaluate how the index structure described in section 4.5 behaves under certain compression levels and tracing limits. The wavelet index is only used to select candidates for the DTW-driven nearest neighbor search. It is not used to provide the actual time series data. This information is retrieved from the uncompressed data set. We doubt that our compression method is good enough to drive both components at the same time, especially when the same maximum error is used.

We set up the following experiment: We use the *n*-grams “democraci”, “drug”, “german” “happi”, “health”, “know”, “money”, “religion”, “soft”, and “war” and a warping radius of 0, 5, 10, 15, and 20. No weight-based filtering is applied. We then compare the top 20 neighbors for all combinations under the set of compression levels and maximum trace-down limits. We compare the number of new entries within that list. If the resulting is shorted than 20, because not enough candidates could be extracted from the index, we do a fill-up with virtual entries that never match any existing one. Without that special handling, only the precision of the index would be measured and low recall values would not affect the score. Also, we do not measure the string distance here since our approach does not influence the pair-wise ranking of the correctly drawn neighbors. This is due to the fact that all candidates returned by the index are compared via DTW without any priority handling.

The results are shown in Figure 6.3. The top graphics shows the index under a maximum compression error of 1, the middle one with an error of 2 and the bottom one with a value of 5. The different plotting styles represent different maximum tracing levels. Sets of markers that share the same x-coordinate usually have their origin in the same warping radius since our tracing algorithm does not take this variable into account. The different groups of markers then correspond to different queries. Measurements that belong to the same query and warping radius, hence the same information request, are connected with a thin line to simplify tracking of the parameter response. It can be observed that different queries are affected differently by the index while the influence of different warping radius values is not that big. Another aspect is the choice of the right compression level. A maximum error of 1 makes it hard to draw enough candidates from the index while an error of 5 results in too many results and therefore does not provide enough value. We therefore assume that a maximum error of around 2 results in good results and we will use this index for further measurements.

Sometimes using the compression rate as hint how exact the nearest neighbor search should be might not be sufficient, for example if the index is already created. Luckily our generic tracer approach allows us to incorporate weight-based filtering. We used that to retrieve data from the index with a maximum error of 2 faster in trade-off against the quality of the results. How the minimum weight threshold affects the quality is shown in Figure 6.4. The setup is identical to Figure 6.3. It turns out that the impact of the filter varies heavily from query to query so it may be hard for users to guess the right value for their expected results. Also, the filter response is not monotonic in neither result quality nor number of executed DTWs. Further research is required to make the filter results more predictable.

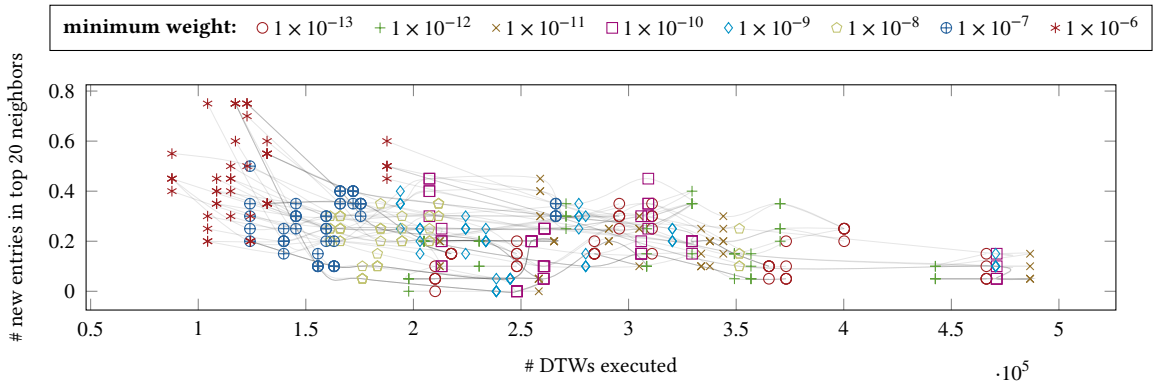

Figure 6.4.: Efficiency of traceback-based indexing, testing minimum weight filter

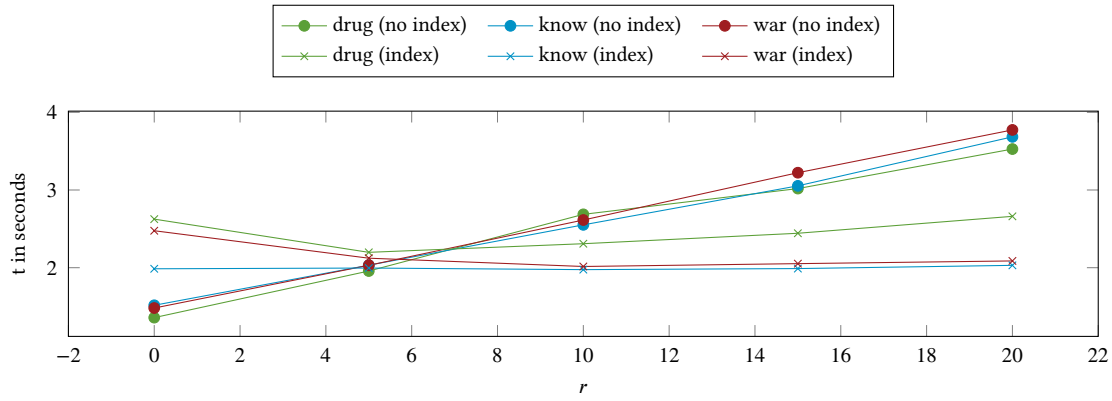

Figure 6.5.: Time of baseline algorithms over r

6.4. Performance

Finally, we want to figure out how fast our algorithm is compared to the baseline. To do so, we carry out the following measurements: On a system described in Table 6.6, we carry out 20 executions and measure the average real time of them. We decided against measuring only user space time because that would miss out IO operations of our memory mapped data storage system.

The first tests apply to the baseline. As shown in Figure 6.5, the runtime of naïve approach grows with increasing values of r. The index-based method keeps this growing at a minimum but starts at a higher initial cost. The break-even point is at around $r = 5$, slightly depending on the query. Also, keep in mind that every warping radius needs its own prepared index. We want to point out that every run has its initial boot-up time for loading the data, even when most data is stored in memory mapped files. But that costs should be the same, no matter what type of query is run and what acceleration method is used.

The next tests belong to our wavelet-based index. Keep in mind that this index does not provide accurate results as shown in the previous section and that the wavelet structure is only used for candidate selection, not for extracting the actual time series data. We

paremeter	value
CPU	Intel® Core™ i5-5200U at 2.2 GHz
RAM	2 times 4 GB DDR3 at 1600 MHz
SSD	Samsung® MZNLN256, 256 GB
CPU Microcode	`0x22`
OS	Arch Linux
Kernel	Linux `4.6.3`
Compiler	Clang `3.8.0`
Linker	GNU gold `1.11`
C-lib	glibc `2.23`
C++-lib	libstdc++ `6.1.1`
Filesystem	Btrfs with `rw, relatime, ssd, space_cache, discard, commit=60, autodefrag, compress=lzo`
Relevant compiler parameters	`-fPIC -fdata-sections -ffunction-sections -g -fno-omit-frame-pointer -mavx2 -mfma -fsized-deallocation -std=c++14 -ffast-math -pthread -O3 -fuse-ld=gold -Wl,--disable-new-dtags -static -Wl,--gc-sections -Wl,--no-export-dynamic`

Table 6.6.: Performance testing environment

selected a subset of possible configurations, namely a maximum tracing level of 4 with no minimum weight filter and a maximum level of 5 with a filter of 0 (no filter), 1×10^{-7}, and 1×10^{-13}. The results are presented in Figure 6.6. We can report that we can accelerate the DTW queries if missing neighbors are an acceptable solution. This can be the case if someone wants only a peek at the data or if algorithms can handle missing links, for example during graph processing.

These results are, to be honest, not convincing when using the algorithm for a general-purpose user-facing application but they might be useful for special algorithms and use-cases.

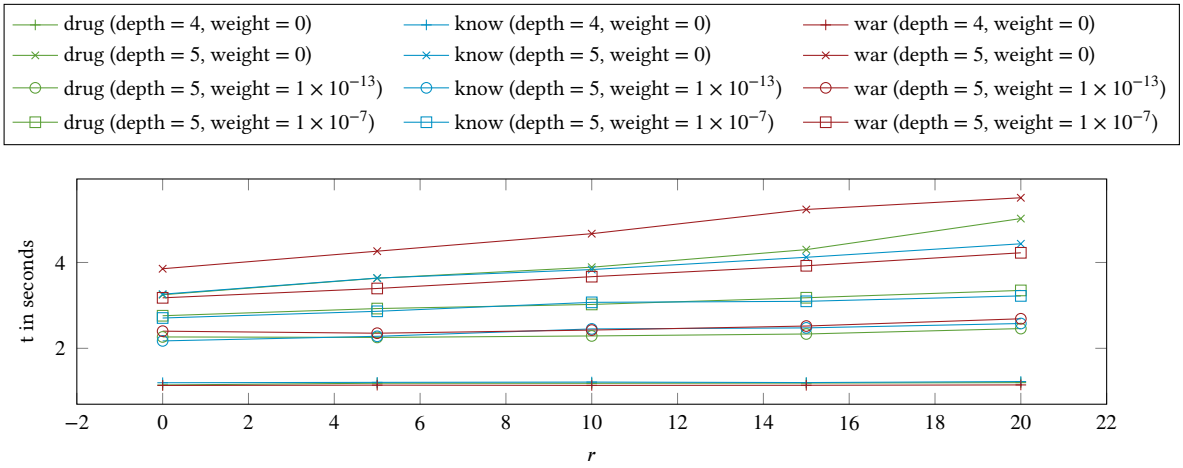

Figure 6.6.: Time of wavelet-index algorithms over r

7. Conclusion

After evaluating the baseline and our ideas we now want to conclude our work and give the reader an overview how it can be used.

We designed a good baseline by providing a solid preprocessing pipeline as well as the explanation how our model of similarity is chosen and why this metric is semantically sane. This baseline can be used for evaluation but also to start experimenting with the results. When limiting the research to 1-grams, the nearest neighbors search is even suitable for interactive investigations and the results can be used by researchers of other fields to explore novel relationships. The data gathering and clean-up part can be extended by adding more sophisticated normalization techniques or error correction methods. Please keep in mind that our definition of similarity might differ from others and we are open to a discussion about it as well as new ideas.

Afterwards, we have shown how the curse of dimensionality affects index selection and why this renders known methods ineffective.

We then presented our own approach to solve this problem, which exploits similarities to compress the data and build an index at the same time. Sadly, our greedy method does not have a high enough chance to succeed, even with the tried improvements. On the other hand: now we know, which approach is not working to solve the problem of fast similarity search and compression at the same time. We were able to collect a good amount of data which shows how the system behaves and which results can be expected. Without doubt, further research is required to make the n-gram data set accessible to humans and machines without the need to use super computers. We believe that the approach of merging trees is unlikely to work when relying on a leaf-to-root greedy approach only. Either a completely different strategy is required or another data representation that makes it easier to find shared information between the time series.

We hope that the implementation, that was carefully designed to be reusable, can be used by others to kickstart their projects. It might be even possible to use parts of the code for production systems. When used for research we hope that more code is published which would enable to recombine, test, reproduce and tune results of other groups. The presentation of the design decisions made during the implementation process may be helpful for other researchers, not only during the actual programming part, but also during the algorithm design.

For the future we are looking forward to see more data-driven research in more fields apart from mathematics and computer science as well as in journalism. We think that data sets like the one provided by Google can have a huge impact. The problem with the current situation is that only large companies are able to produce such collections and that copyright restrictions and economic interests are a major issue for science. The availability of a large-scale digital library with a proper API would enable researchers to extract their own n-gram data sets with custom publication weights and pruning methods.

Apart from the concrete of Google data set we have used, our method might work for other data sets. We did not test this but may do in future publications. Especially for a smaller set of time series which have more sample points with lower frequencies the algorithm may result in better results. For this kind of data, the prior work needs to be re-evaluated of course.

A last and short word on the personal gain through this project: We learned a lot about this data set and how to implement and profile performant algorithms in general, how to structure implementations in a way that are reusable and interceptable and as a not unimportant skill how to gather and visualize data for publications as well as how to create explanatory diagrams.

Bibliography

[1] Yuri Lin et al. “Syntactic Annotations for the Google Books Ngram Corpus”. In: *Proceedings of the ACL 2012 System Demonstrations*. ACL ’12. Jeju Island, Korea: Association for Computational Linguistics, 2012, pp. 169–174.

[2] Vivek Kulkarni et al. “Statistically Significant Detection of Linguistic Change”. In: *CoRR* abs/1411.3315 (2014). URL: http://arxiv.org/abs/1411.3315.

[3] Y. W. Chao et al. “Mining semantic affordances of visual object categories”. In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2015, pp. 4259–4267. DOI: 10.1109/CVPR.2015.7299054.

[4] Irem Uz. “Individualism and First Person Pronoun Use in Written Texts Across Languages”. In: *Journal of Cross-Cultural Psychology* 45.10 (2014), pp. 1671–1678. DOI: 10.1177/0022022114550481. eprint: http://jcc.sagepub.com/content/45/10/1671.full.pdf+html. URL: http://jcc.sagepub.com/content/45/10/1671.abstract.

[5] Niveda Krishnamoorthy et al. “Generating Natural-language Video Descriptions Using Text-mined Knowledge”. In: *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence*. AAAI’13. Bellevue, Washington: AAAI Press, 2013, pp. 541–547.

[6] Steven T. Piantadosi. “Zipf’s word frequency law in natural language: A critical review and future directions”. In: *Psychonomic Bulletin & Review* 21.5 (2014), pp. 1112–1130. ISSN: 1531-5320. DOI: 10.3758/s13423-014-0585-6.

[7] Fakhteh Ghanbarnejad et al. “Extracting information from S-curves of language change”. In: *Journal of The Royal Society Interface* 11.101 (2014). ISSN: 1742-5689. DOI: 10.1098/rsif.2014.1044. eprint: http://rsif.royalsocietypublishing.org/content/11/101/20141044.full.pdf. URL: http://rsif.royalsocietypublishing.org/content/11/101/20141044.

[8] Igor Grossmann and Michael E. W. Varnum. “Social Structure, Infectious Diseases, Disasters, Secularism, and Cultural Change in America”. In: *Psychological Science* 26.3 (2015), pp. 311–324. DOI: 10.1177/0956797614563765. eprint: http://pss.sagepub.com/content/26/3/311.full.pdf+html. URL: http://pss.sagepub.com/content/26/3/311.abstract.

[9] Daniel Naber. *Finding errors using Big Data*. 2015. URL: http://wiki.languagetool.org/finding-errors-using-big-data.

[10] Clifford R. Mynatt, Michael E. Doherty, and Ryan D. Tweney. "Confirmation bias in a simulated research environment: An experimental study of scientific inference". In: *Quarterly Journal of Experimental Psychology* 29.1 (1977), pp. 85–95. DOI: 10.1080/00335557743000053. eprint: http://dx.doi.org/10.1080/00335557743000053.

[11] Raymond S. Nickerson. "Confirmation bias: A ubiquitous phenomenon in many guises." In: *Review of General Psychology* 2.2 (1998), pp. 175–220. ISSN: 1939-1552 (Electronic); 1089-2680 (Print). DOI: 10.1037/1089-2680.2.2.175.

[12] Eva Jonas et al. "Confirmation bias in sequential information search after preliminary decisions: An expansion of dissonance theoretical research on selective exposure to information." In: *Journal of Personality and Social Psychology* 80.4 (2001), pp. 557–571. ISSN: 1939-1315 (Electronic); 0022-3514 (Print). DOI: 10.1037/0022-3514.80.4.557.

[13] Richard A. White Jeffrey J. McMillan. "Auditors' Belief Revisions and Evidence Search: The Effect of Hypothesis Frame, Confirmation Bias, and Professional Skepticism". In: *The Accounting Review* 68.3 (1993), pp. 443–465. ISSN: 00014826.

[14] P. C. Wason. "On the failure to eliminate hypotheses in a conceptual task." In: *The Quarterly Journal of Experimental Psychology* 12 (1960), pp. 129–140. ISSN: 0033-555X(Print). DOI: 10.1080/17470216008416717.

[15] Eamonn Keogh and Ann Chotirat Ratanamahatana. "Exact indexing of dynamic time warping". In: *Knowledge and Information Systems* 7.3 (2005), pp. 358–386. ISSN: 0219-3116. DOI: 10.1007/s10115-004-0154-9.

[16] Eitan Adam Pechenick, Christopher M. Danforth, and Peter Sheridan Dodds. "Characterizing the Google Books Corpus: Strong Limits to Inferences of Socio-Cultural and Linguistic Evolution". In: *PLoS ONE* 10.10 (Oct. 2015), pp. 1–24. DOI: 10.1371/journal.pone.0137041.

[17] Daniel Lemire. "Faster Retrieval with a Two-Pass Dynamic-Time-Warping Lower Bound". In: *CoRR* abs/0811.3301 (2008). URL: http://arxiv.org/abs/0811.3301.

[18] Eamonn Keogh et al. "Dimensionality Reduction for Fast Similarity Search in Large Time Series Databases". In: *Knowledge and Information Systems* 3.3 (2001), pp. 263–286. ISSN: 0219-1377. DOI: 10.1007/PL00011669.

[19] Kaushik Chakrabarti et al. "Locally Adaptive Dimensionality Reduction for Indexing Large Time Series Databases". In: *ACM Trans. Database Syst.* 27.2 (June 2002), pp. 188–228. ISSN: 0362-5915. DOI: 10.1145/568518.568520.

[20] Kin-Pong Chan and Ada Wai-Chee Fu. "Efficient time series matching by wavelets". In: *Data Engineering, 1999. Proceedings., 15th International Conference on*. Mar. 1999, pp. 126–133. DOI: 10.1109/ICDE.1999.754915.

[21] Jin Shieh and Eamonn Keogh. "iSAX: Indexing and Mining Terabyte Sized Time Series". In: *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '08. Las Vegas, Nevada, USA: ACM, 2008, pp. 623–631. ISBN: 978-1-60558-193-4. DOI: 10.1145/1401890.1401966.

[22] Eugene Fink and Harith Suman Gandhi. "Compression of Time Series by Extracting Major Extrema". In: *J. Exp. Theor. Artif. Intell.* 23.2 (June 2011), pp. 255–270. ISSN: 0952-813X. DOI: 10.1080/0952813X.2010.505800.

[23] Fabian Mörchen. *Time series feature extraction for data mining using DWT and DFT*. Tech. rep. 2003.

[24] A. Jovic and N. Bogunovic. "Feature Extraction for ECG Time-Series Mining Based on Chaos Theory". In: *2007 29th International Conference on Information Technology Interfaces*. June 2007, pp. 63–68. DOI: 10.1109/ITI.2007.4283745.

[25] Hui Zhang et al. "Unsupervised feature extraction for time series clustering using orthogonal wavelet transform". In: *Informatica* 30.3 (2006).

[26] R. J. Alcock et al. "Time-series similarity queries employing a feature-based approach". In: *In 7 th Hellenic Conference on Informatics, Ioannina*. 1999, pp. 27–29.

[27] Fabian Keller, Emmanuel Müller, and Klemens Böhm. "Estimating Mutual Information on Data Streams". In: *Proceedings of the 27th International Conference on Scientific and Statistical Database Management*. SSDBM '15. La Jolla, California: ACM, 2015, 3:1–3:12. ISBN: 978-1-4503-3709-0. DOI: 10.1145/2791347.2791348.

[28] Rakesh Agrawal et al. "Fast Similarity Search in the Presence of Noise, Scaling, and Translation in Time-Series Databases". In: *Proceedings of the 21th International Conference on Very Large Data Bases*. VLDB '95. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995, pp. 490–501. ISBN: 1-55860-379-4.

[29] Gautam Das et al. "Rule Discovery from Time Series." In: *KDD*. Vol. 98. 1. 1998, pp. 16–22.

[30] Michael D. Morse and Jignesh M. Patel. "An Efficient and Accurate Method for Evaluating Time Series Similarity". In: *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*. SIGMOD '07. Beijing, China: ACM, 2007, pp. 569–580. ISBN: 978-1-59593-686-8. DOI: 10.1145/1247480.1247544.

[31] N. Viovy, O. Arino, and A. S. Belward. "The Best Index Slope Extraction (BISE): A method for reducing noise in NDVI time-series". In: *International Journal of Remote Sensing* 13 (May 1992), pp. 1585–1590. DOI: 10.1080/01431169208904212.

[32] Mark Davis and Ken Whistler. *Unicode Standard Annex #15: Unicode Normalization Forms*. 2015. URL: http://unicode.org/reports/tr15/.

[33] George A. Miller. "WordNet: A Lexical Database for English". In: *Commun. ACM* 38.11 (Nov. 1995), pp. 39–41. ISSN: 0001-0782. DOI: 10.1145/219717.219748.

[34] Martin Porter. *Developing the English stemmer*. 2002. URL: http://snowball.tartarus.org/algorithms/english/stemmer.html.

[35] Peder Olesen Larsen and Markus von Ins. "The rate of growth in scientific publication and the decline in coverage provided by Science Citation Index". In: *Scientometrics* 84.3 (2010), pp. 575–603. ISSN: 1588-2861. DOI: 10.1007/s11192-010-0202-z.

[36] Lutz Bornmann and Rüdiger Mutz. "Growth rates of modern science: A bibliometric analysis". In: *CoRR* abs/1402.4578 (2014). URL: http://arxiv.org/abs/1402.4578.

[37] Michael Feindt. "A Neural Bayesian Estimator for Conditional Probability Densities". In: (Feb. 2004). URL: https://arxiv.org/abs/physics/0402093.

[38] M. E. Munich and P. Perona. "Continuous dynamic time warping for translation-invariant curve alignment with applications to signature verification". In: *Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on*. Vol. 1. 1999, 108–115 vol.1. DOI: 10.1109/ICCV.1999.791205.

[39] Paul H. C. Eilers*. "Parametric Time Warping". In: *Analytical Chemistry* 76.2 (2004), pp. 404–411. DOI: 10.1021/ac034800e. eprint: http://dx.doi.org/10.1021/ac034800e.

[40] Joseph B Kruskal and Mark Liberman. "The symmetric time-warping problem: from continuous to discrete". In: *Time warps, string edits and macromolecules: The theory and practice of sequence comparison* (1983), pp. 125–161.

[41] Eamonn J. Keogh and Michael J. Pazzani. "Derivative Dynamic Time Warping". In: *In First SIAM International Conference on Data Mining (SDM'2001)*. 2001.

[42] Hiroaki Sakoe and Seibi Chiba. "Dynamic programming algorithm optimization for spoken word recognition". In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 1 (1978), pp. 43–49.

[43] Antonin Guttman. "R-trees: A Dynamic Index Structure for Spatial Searching". In: *SIGMOD Rec.* 14.2 (June 1984), pp. 47–57. ISSN: 0163-5808. DOI: 10.1145/971697.602266.

[44] Thomas Seidl and Hans-Peter Kriegel. "Optimal Multi-step K-nearest Neighbor Search". In: *SIGMOD Rec.* 27.2 (June 1998), pp. 154–165. ISSN: 0163-5808. DOI: 10.1145/276305.276319.

[45] N. Ahmed, T. Natarajan, and K. R. Rao. "Discrete Cosine Transform". In: *IEEE Transactions on Computers* C-23.1 (Jan. 1974), pp. 90–93. ISSN: 0018-9340. DOI: 10.1109/T-C.1974.223784.

[46] K. R. Rao and P. Yip. *Discrete Cosine Transform: Algorithms, Advantages, Applications*. San Diego, CA, USA: Academic Press Professional, Inc., 1990. ISBN: 0-12-580203-X.

[47] Alfred Haar. "Zur Theorie der orthogonalen Funktionensysteme". In: *Mathematische Annalen* 69.3 (1910), pp. 331–371. ISSN: 1432-1807. DOI: 10.1007/BF01456326.

[48] Karen Egiazarian and Jaakko Astola. "Tree-Structured Haar Transforms". In: *Journal of Mathematical Imaging and Vision* 16.3 (2002), pp. 269–279. ISSN: 1573-7683. DOI: 10.1023/A:1020385811959.

[49] Huamin Chen, Jian Li, and P. Mohapatra. "RACE: time series compression with rate adaptivity and error bound for sensor networks". In: *Mobile Ad-hoc and Sensor Systems, 2004 IEEE International Conference on*. Oct. 2004, pp. 124–133. DOI: 10.1109/MAHSS.2004.1392089.

[50] "IEEE Standard for Floating-Point Arithmetic". In: *IEEE Std 754-2008* (Aug. 2008), pp. 1–70. DOI: 10.1109/IEEESTD.2008.4610935.

[51] Jack Rae Marius Muja David G. Lowe. *FLANN - Fast Library for Approximate Nearest Neighbors*. Version 1.8.4. URL: http://www.cs.ubc.ca/research/flann/.

[52] Ingrid Daubechies. *Ten Lectures on Wavelets (CBMS-NSF Regional Conference Series in Applied Mathematics)*. SIAM: Society for Industrial and Applied Mathematics, 1992. ISBN: 0898712742.

[53] *G.711: Pulse code modulation (PCM) of voice frequencies*. Geneva, Switzerland, Nov. 1988. URL: https://www.itu.int/rec/T-REC-G.711.

[54] *ISO/IEC 14882:2014*. Tech. rep. International Organization for Standardization, 2014.

[55] Scott Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*. 3rd Edition. Addison-Wesley Professional, 2005. ISBN: 978-0321334879.

[56] Scott Meyers. *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14*. O'Reilly Media, 2014. ISBN: 1-4919-0398-8.

[57] Tyler Vigen. *Spurious Correlations*. Hachette Books, 2015. ISBN: 978-0316339438.

[58] Владимир Иосифович Левенштейн. "Двоичные коды с исправлением выпадений, вставок и замещений символов". In: *Доклады Академий Наук СССР* 163.4 (1965), pp. 845–848.

A. Appendix

letter	raw	filtered	s.normalized	w.normalized	pruned
a	1 440 378	493 706	361 035	276 583	52 766
b	1 018 343	377 220	281 903	227 025	48 380
c	1 546 639	532 497	380 823	288 078	59 439
d	981 466	341 819	249 790	188 481	38 220
e	829 512	288 878	213 871	155 833	29 578
f	742 479	251 520	182 495	141 276	27 290
g	675 417	250 442	187 863	152 828	32 088
h	763 425	271 203	199 303	160 263	33 337
i	847 619	268 572	191 304	136 416	23 698
j	276 339	99 430	70 257	58 083	11 241
k	536 396	222 419	172 642	150 450	30 335
l	779 711	266 197	194 019	155 416	31 561
m	1 231 265	448 398	327 496	261 578	56 082
n	642 961	223 966	163 531	128 920	26 837
o	606 102	209 806	156 250	118 133	20 383
p	1 464 402	518 315	373 821	286 958	57 440
q	108 085	37 278	26 777	21 695	4124
r	886 734	302 916	223 977	167 128	32 791
s	1 834 759	653 305	478 905	378 824	77 857
t	1 106 849	384 978	275 879	219 654	42 432
u	357 527	130 599	96 580	69 671	13 725
v	451 224	166 432	122 916	100 387	19 142
w	511 180	159 192	117 519	96 570	18 893
x	66 100	20 082	12 810	11 384	2222
y	127 204	43 844	31 780	27 932	5548
z	138 100	55 448	43 393	38 363	7642
$\sum$	19 970 216	7 018 462	5 136 939	4 017 929	803 051

Table A.1.: Total amount of 1-grams after each clean-up step

letter	raw	filtered	s.normalized	w.normalized	pruned
a	36 328 811	7 163 539	5 618 464	3 437 351	569 502
b	22 289 608	4 306 283	3 474 074	2 107 814	313 798
c	35 498 239	6 958 880	5 631 223	3 050 993	480 126
d	21 801 951	4 257 630	3 514 067	1 962 782	289 009
e	19 115 156	3 788 663	3 128 430	1 731 101	271 638
f	20 294 666	4 030 176	3 219 551	1 860 448	306 162
g	12 302 512	2 327 082	1 905 063	1 132 058	162 562
h	16 144 346	3 121 167	2 521 984	1 522 944	230 591
i	20 944 241	4 188 717	3 300 899	1 815 712	308 527
j	4 871 485	852 751	720 116	508 313	65 061
k	4 447 999	679 656	578 279	399 545	41 193
l	16 610 230	3 139 429	2 529 904	1 513 381	233 107
m	24 450 309	4 616 893	3 738 647	2 244 811	335 992
n	12 149 710	2 229 459	1 769 874	1 130 139	164 102
o	17 346 280	3 474 432	2 687 935	1 646 478	300 960
p	30 278 927	5 813 503	4 758 716	2 564 546	399 868
q	1 870 891	352 107	294 104	189 536	23 331
r	19 648 081	3 882 421	3 222 234	1 676 035	265 316
s	42 556 368	8 478 842	6 824 045	3 846 241	619 708
t	31 880 903	6 379 694	4 728 150	2 929 660	525 358
u	7 989 152	1 617 535	1 370 469	862 065	108 033
v	7 333 265	1 289 810	1 056 425	673 484	90 250
w	15 825 201	3 223 549	2 494 916	1 514 552	264 322
x	572 086	67 994	43 602	36 484	3755
y	2 727 242	517 629	382 733	248 245	41 013
z	943 511	126 500	106 298	85 216	7178
$\sum$	446 221 170	86 884 341	69 620 202	40 689 934	6 420 462

Table A.2.: Total amount of 2-grams after each clean-up step

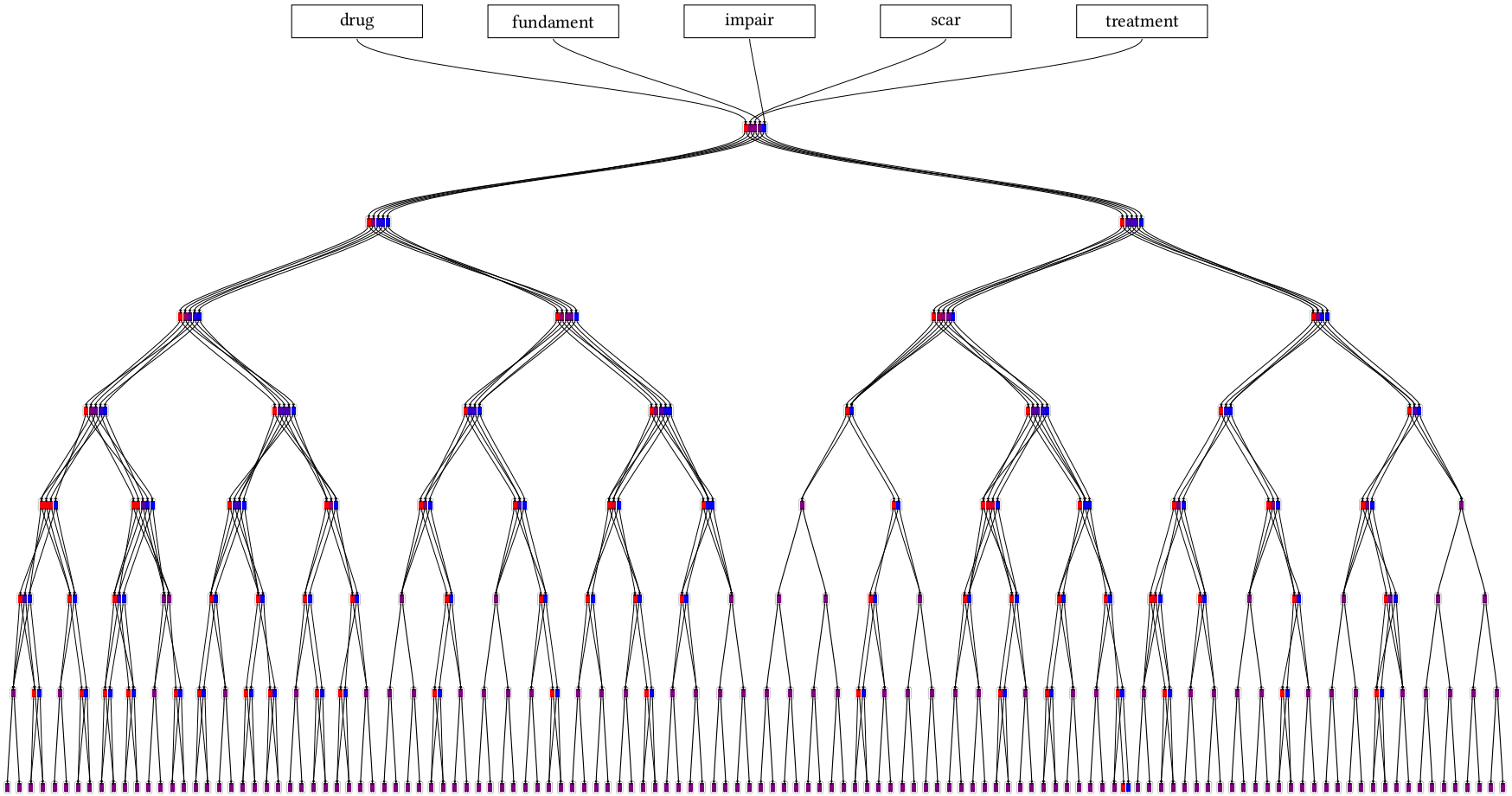

Figure A.1.: Example tree merges, nearest neighbors of “drug” in $[1, 256]$ with $r = 10$

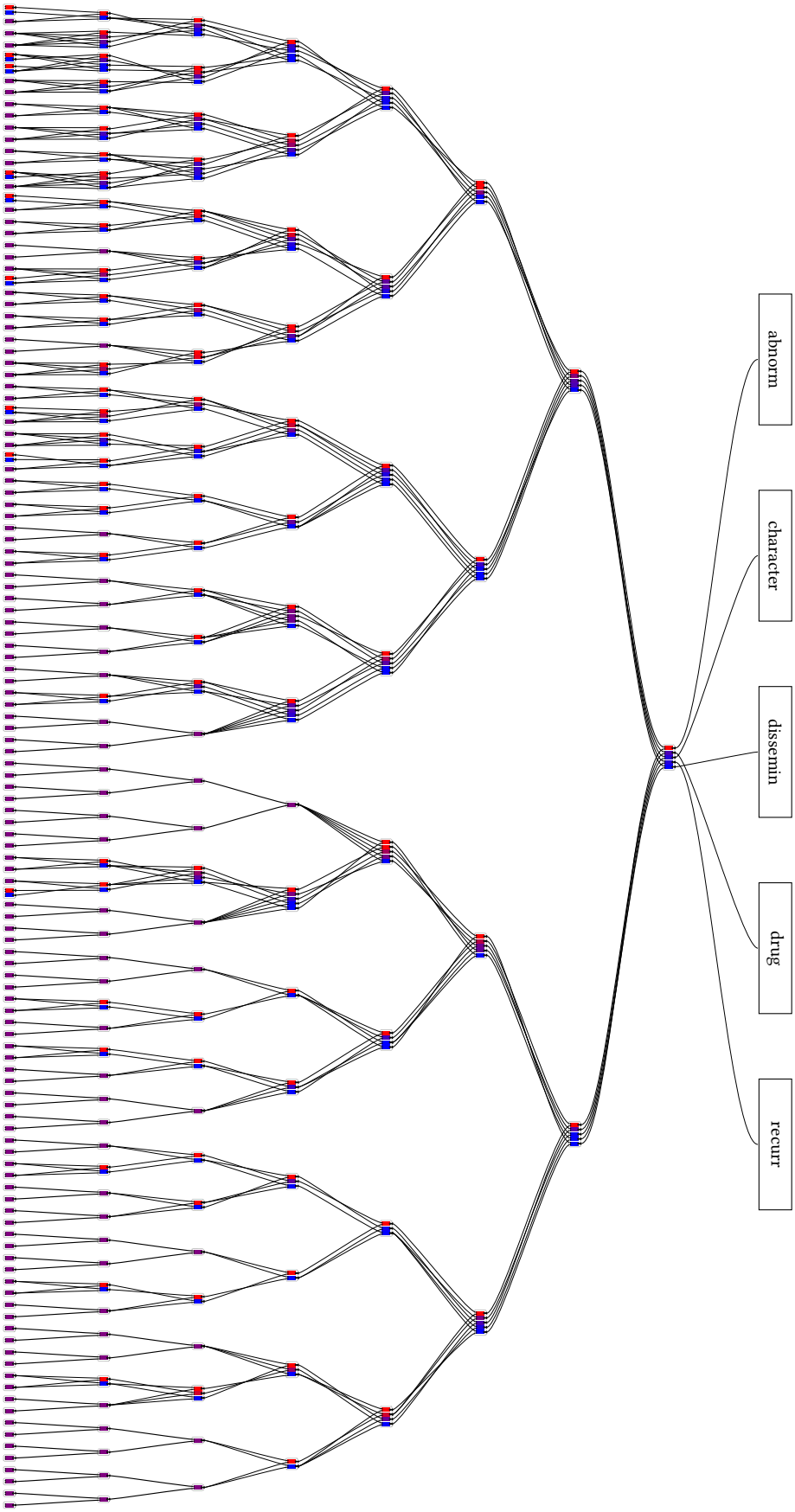

Figure A.2.: Example tree merges, nearest neighbors of "drug" in [129, 256] with $r = 10$

library	usage	license	version
Boost	align (aligned memory allocation), functional (hash calculation), geometry (R-tree index), interprocess (managed mapped file), iostream (mmaped file), locale (UTF conversion), program option parser	Boost Software License — Version 1.0	`1.60.0`
half	half-precision IEEE floating point emulation	MIT License	`1.11.0`
int48_t	emulation of signed 48-bit integer	MIT License	`GIT:ccfd6891`
ICU	Unicode handling	ICU License — ICU 1.8.1 and later	`57.1`
libsimdpp	high-level interface to SSE/AVX/FMA/…	Boost Software License — Version 1.0	`2.0-rc2`
tcmalloc	fast replacement for libc malloc	BSD 3-Clause License	`2.5` (part of gperftools)
wavelib	Discrete Wavelet Transform	BSD 3-Clause License (includes modifiction)	`GIT:a2c70971`

Table A.3.: Used libraries during the implementation