GEORG-AUGUST-UNIVERSITÄT
GÖTTINGEN

Bachelor's Thesis

submitted in partial fulfillment of the
requirements for the degree of B.Sc in
"Applied Computer Science"

Learning Patterns from Textual Data

Mareike Burg

Institute of Computer Science

Bachelor's and Master's Theses
of the Center for Computational Sciences
at the Georg-August-Universität Göttingen

02. September 2021

Georg-August-Universität Göttingen
Institute of Computer Science

Goldschmidtstraße 7
37077 Göttingen
Germany

☎ +49 (551) 39-172000
FAX +49 (551) 39-14403
✉ office@informatik.uni-goettingen.de
🌍 www.informatik.uni-goettingen.de

First Supervisor:	Prof. Dr. Manea
Second Supervisor:	Prof. Dr. Damm

I hereby declare that I have written this thesis independently without any help from others and without the use of documents or aids other than those stated. I have mentioned all used sources and cited them correctly according to established academic citation rules.

Göttingen, 02. September 2021

Mareike Burg

Contents

List of Listings

Chapter 1

Introduction

1.1 Motivation

Words and texts are an essential part of human civilisation, they are used to communicated with each other, store information or build huge databases to search for new correlations. Most of these texts are part of a language, so they have a direct meaning to us, while others represent different types of information. That could be a list of names, columns of numbers or even a binary representation of a video file. All data that can be stored as a text can be called **textual data**.

Their widespread appearance makes them an important topic in computer science, it should be easy and efficient to store textual data, and most programming languages have an own datatype called *Strings* to handle textual data. But data needs not only to be stored, it should also be possible to work with it, be it searching for specific elements or performing operations on it, like merging two data sets together. These techniques are not only improved by practical optimizations, but also include problems in theoretical computer science, solving them could yield immense performance.

One interesting area is the concept of **patterns**, a pattern is used to describe the similarity between multiple words. An example for that would be the three words abc, $abbc$ and $abbbc$, they all start with the letter a, end with the letter c and have a variable amount of b's in the middle. This allows to see whether another word has the same pattern or not, the word $abbbbc$ looks similar to the others, while the word xyz looks really different and does not fit the same pattern. **Learning** such a pattern reveals information about the original words, and enables finding other words that share the same pattern as this example. It can also be used directly to solve the problem of regular expressions defined by Kleene in [1], for more about regular expressions see the work by Campeanu et al. [2] and Freydenberger [3].

But the pattern can not only be a single letter that is at the same position in all words, it could also be a repeating sequence. The sequence could be different for each word, but it could be that it appears always at a fixed positions in all words.

Furthermore, a pattern can not only be found for two words, it can also be learned from multiple sentences or complete separate data sets. An application of this is the analysis of genetic codes, they can be translated into a text build from four different symbols and is very similar over large parts for most organisms. This makes the subtle differences even more relevant, for example when different species are related to each other, but the exact lineage is unknown. If a pattern is found that matches some species of a bigger class of species, but not others , an indication for which of the species are related is given.

1.2 Goal of work

This thesis focuses on two tasks around patterns, the matching problem and the problem of finding a descriptive pattern Desc-Pat. The **matching problem** is the decision problem of checking whether a given word and a given pattern match together. Second, the **Desc-Pat** problem is about finding a pattern for a given sample of words, importantly must the pattern be descriptive of the sample, which means that the pattern describes the sample as accurate as possible, and only matches a minimal amount of words that are not from the sample. Finding such a descriptive pattern efficiently allows to solve the above defined problems.
The Desc-Pat problem requires a solution for the matching problem, and since the matching problem is NP-Complete in general, Desc-Pat is aswell. But as shown by Fernau et al. [4] it is possible to define the matching problem for patterns of one pattern class, where an efficient algorithm for different pattern classes is possible. This algorithm combined with the one defined by Fernau et al. [5] allows to **learn** a descriptive pattern from a sample in polynomial time.

The main purpose of this work is to implement the algorithms for the **matching problem** and **Desc-Pat** problem for the pattern classes of regular patterns and for one repeated patterns. Furthermore, the performance of the algorithms needs to be tested to see whether they are viable for practical use in their current state. While at it, it can be compared how big the differences between two generated patterns are, if different pattern classes are used on the same data. This includes measuring whether a learned descriptive pattern is useful. Furthermore, it is to be tested, how effective the **Desc-Pat** algorithm is in confirming if other words belong to its sample or not.
Last but not least, is attempted to use the algorithm of learning patterns to find how closely subspecies from one class of species are related.

1.3 Thesis Structure

First, the formal definition of patterns, the matching and the learning problem are defined in section 2. This includes the notations that are used in this work and a paragraph on **pattern matching**, since the concept is needed later. The next section 3 explains how the algorithms work, how the data will be generated and which measurement techniques will be used to rate the output from the algorithms.
Subsequently, will this be implemented in section 4, there is described what the thought process behind implementation was, and how the implementation was structured. In chapter 5 are the results of the test cases bee shown, while the conclusion and the analysis of problems are found in chapter 6.

Chapter 2

Basics

2.1 Words and Patterns

Formally defined, a *word* is any sequence of symbols, and all *symbols* are part of an *alphabet* Σ. $|\Sigma|$ is the *number of symbols* in the alphabet, Σ^* is the set of all words that can be generated from the alphabet Σ, whereas Σ^+ refers to the set of non-empty words of the alphabet. ε is the empty word, it contains no symbol and has length 0. It should be noted that any punctuation symbol like the period ".", or the space character " " is also a *symbol*, therefore can any sentence or text be stored in one *word*.
So far, all symbols only represent a single character, they are called *terminals*. In contrast to that, *variables* do not represent one fixed character, they can be replaced by *words*. There are endless replacement possibilities for each variable every replacement is described by a function that maps a *variable* A to a *word* w that only contains terminals. A variable could also be replaced by another variable, or a mix of variables and terminals, but this will not be considered in the context of this work. Furthermore, in continuity with the definitions by Fernau et al. [4], a variable can not be replaced by the empty word ε.

The set of all variables is called X, and no symbol can be a terminal and variable at the same time, therefore $\Sigma \cap X = \emptyset$. With X and Σ defined, a replacement function for each variable $A \in X$ is defined as $f : A \to w \in \Sigma^+$. To always have a clear differentiation between variables and terminals, variables are always written as upper case letters, while terminals are only lower case letters.

This can be seen in the example below, where a set of variables X, a possible alphabet Σ and a word w_1 from Σ^* is defined.

$$X = A, B$$
$$\Sigma = a, b, c, z$$
$$w_1 = zabcdz$$

As defined by Angluin [6], *variables* and *terminals* can be combined to form a *pattern* $\alpha \in (\Sigma \cup X)^*$. The set of different terminal symbols occurring in a pattern α is called $term(\alpha)$, and the set of variables is named $var(\alpha)$. For each symbol $\triangle \in \Sigma \cup X$ is $|\alpha|_\triangle$ the number of times this symbol appears in the pattern. Variables $A \in X$ with $|\alpha|_A \geq 2$ are called *repeated* variables and the set of all repeating variables in α is called $rvar(\alpha)$.

By using a replacement functions f for each variable in a pattern α, the complete pattern can be mapped to a word only consisting of terminals. Such a function is called $h : \alpha \to w \in \Sigma^+$, and a small example can be seen below, where a word is generated from a pattern.

$$\alpha = zABz$$
$$h_1 : A \to ab,\ B \to cd$$
$$h_1(\alpha) = zabcdz$$

2.2 Matching Problem

By applying a suitable function h to a pattern α, a word w can be generated from α. By reversing this, it can be checked whether a word can be generated from a pattern, this is called *matching*. α *matches* w when a function h exists, such that $h(\alpha) = w$. If no such h exists, the pattern does not match the word, therefore no matter how the variables are replaced, the word w can not be generated. Since variables can not be replaced by ε, a word w cannot be matched by a pattern α, when $|\alpha| < |w|$.
The set of all words that is matched by α is called the *pattern language* of α, its abbreviation is $L(\alpha)$. Having to decide, whether a given word w is matched by a fixed pattern, is called the *Matching Problem*, which can also be described by $w \in L(\alpha)$.

The word w_1 defined above is matched by the α from the last page, while the newly defined w_2 from the next page is not, since the first terminal of the α is always a z and w_2 starts with an a.

$$w_1 = zabcdz \in L(\alpha)$$
$$w_2 = azaz \notin L(\alpha)$$

As proven by Angluin [6], the matching problem is for all patterns *NP-complete*, so no efficient algorithms for solving the matching problem is currently possible. But there are patterns with specific properties that can be solved in polynomial time, therefore patterns with those properties are grouped into *pattern classes*. Finding if a given pattern is part of a pattern class is called the *membership problem*. Two pattern classes and their respective algorithms to solve the matching problem in polynomial time are explained below, as they were defined by Fernau et al. [4].

2.2.1 Regular Patterns

A pattern is *regular* if there are no repeated variables in the pattern. Each variable that occurs once in the pattern is not allowed to appear again in the pattern, this property can be described as α is regular if $\forall A \in var(\alpha) : |\alpha|_A = 1$. The matching problem for regular patterns is solved in linear time using the algorithm explained in section 3.1. β is an example for a regular patterns, since its variables A and B appear only one time.

$$\beta = aAbB$$

2.2.2 One repeated Patterns

The definition of regular patterns is specific, each variable may occurs exactly once. The *one repeated patterns* extend regular patterns by allowing one variable to repeat itself. So one special variable can occur multiple times, while the other variables are still limited to appear once. There exists an algorithm to solve the matching problem in quadratic time, but only a weaker variant of the algorithm with cubic runtime is implemented in section 3.2, both variants are defined by Fernau et al. [4].

In the following example, δ is a one repeated pattern, which also matches the word $w_3 = ccbccbcc$.

$$\delta = AbAbA$$
$$h_2 : A \to cc$$
$$h_2(\delta) = ccbccbcc = w_3$$

2.3 Learning Patterns

The matching problem is not only important for a single word, but it can also be applied on a set of words at the same time. Such a finite set of words is called a *sample* S, and the task to find a pattern α that matches all word from S is called *learning a pattern* from the sample. This is done by finding an α, with its respective pattern language $L(\alpha)$, that every word $w \in S$ is in $L(\alpha)$, therefore $S \subseteq L(\alpha)$.

However, this is a weak definition, since the pattern $\zeta = A$ with only one variable can match every single word except ε, it could be generated for any sample. The key point is the size of the pattern language $L(\alpha)$, as it decides how well a learned pattern fits to a sample. If it is too big, like in $L(\zeta)$, many words that are not similar to the original sample would be matched by the pattern, making the pattern *unspecific*. On the other hand, a pattern could also not match all words from a sample, so it would be *to specific*.
This allows the definition of an ideal pattern for a sample, the *descriptive pattern*. A pattern α is *descriptive* for a sample S, if there is no other pattern β that also matches all words from the sample and has a smaller pattern language, so $\nexists\, \beta$ with $S \subseteq L(\beta) \subset L(\alpha)$. In the example below, the patter η is more specif then θ, since all words that can be generated from η start with the terminal a. θ on the other hand can start with any terminal, which is why the equation 2.1 hold true, and θ not descriptive for S is. It can be shown that η is descriptive for S.

$$S = abc,\ abbc,\ abbbc$$
$$\eta = aBc,\ S \subset L(\eta)$$
$$\theta = ABc,\ S \subset L(\theta)$$
$$S \subseteq L(\eta) \subset L(\theta) \tag{2.1}$$

The definition of *descriptive patterns* does not imply that there is only one descriptive patterns for a sample, there can be multiple patterns that are descriptive for one sample. The following example shows another descriptive pattern ι for the previously defined sample S. $L(\iota)$ is not a subset of $L(\eta)$, since the word *bbc* is not in $L(\eta)$, and on the other side, $L(\eta)$ is also not a subset of $L(\iota)$, since acc is not in $L(\iota)$. Therefore, both patterns can be descriptive for S, if no other pattern exists whose pattern language is a subset of $L(\eta)$ and $L(\iota)$.

$$\iota = Abc$$
$$S \subseteq L(\iota),\ S \subseteq L(\eta)$$
$$L(\iota) \not\subset L(\eta),\ L(\eta) \not\subset L(\iota)$$

The problem of finding a descriptive pattern for a sample is called *Desc-Pat*, and finding such a pattern depends on knowing that all words from the sample are matched by the found pattern. Subsequently, to solve *Desc-Pat* in polynomial time, an algorithm to solve the matching problem in polynomial time is needed. Therefore, the problem Π-*Desc-Pat* for a class of pattern Π is defined. By using a class of patterns Π, like the class of regular pattern, where the matching problem can be solved efficiently, Π-*Desc-Pat* is also be solved efficiently and a descriptive pattern from Π can be found. This is done by Fernau et al. [5] and that algorithm is further explained in section 3.3.

2.4 Additional Definitions and Notations

The upcoming sections contains definitions and notations that are used later. The *length* of a pattern α is defined as $|\alpha|$, it describes the total number of symbols. A single symbol at the position i in α is referred as $\alpha[i]$, with i being a natural in the range from 0 to length of $\alpha - 1$. Consequently is $\alpha[0]$ the first symbol of a pattern, while $\alpha[|\alpha| - 1]$ is the last one. This is extended to refer to multiple symbols in a row, $\alpha[i..j]$ is defined as all symbols from position i up to j, as long as $i \leq j$ and $0 \leq i, j \leq |\alpha| - 1$. Two Patterns α and β can be concatenated by $\alpha\beta$, for multiple patterns α_1, α_2, .., α_j, the notation with the product symbol $\prod_{\text{i=1,k}} \alpha_i$ is used.
Any part γ of a non-empty pattern α is called a *factor*, if α can be split into $\alpha = \beta\gamma\delta$ with $\beta, \gamma, \delta \in (\Sigma \cup X)^*$. If β is empty, then the factor γ is also called a *prefix* of α, similarly, if δ is empty, the factor γ is named a *suffix* of α.

Since *patterns* are build from *variables* and *terminals*, while *words* are only build from *terminals*, every *word* is also a *pattern*, and the notations for *patterns* can also be used for *words*.

2.4.1 Maximal Terminal Factors

Another property of a pattern α needed later, are the *maximal terminal factors* of α, they are all the parts of α that only contain *terminals*. This is described as the *factors* from α that only contain $terminals$, and are the maximal size, so there is no terminal next to the factor that could be included in the *maximal terminal factors*. Therefore, is $\gamma = \alpha[i..i + |\gamma|]$ a maximal terminal factor of α, if $\alpha[i-1]$ and $\alpha[i + |\gamma| + 1]$ are variables or do not exist, since they are outside of the pattern. The set of all unique *maximal terminal factors* is called *MTF*, and this is shown below for the pattern κ.

$$\kappa = abAcdefBghiCj$$
$$MTF = \{ab, cdef, ghi, j\}$$

2.4.2 Canonical Form

The *canonical form* of a pattern is used to order the variables in a pattern α, this is done by renaming each variables in the pattern . First, the set of variables $var(\alpha)$ is ordered, such that each variable has a ranking from 1 to $|var(\alpha)|$. Then the pattern is swiped from left to right, and all variables are renamed depending on their first appearance. After that, the leftmost occurrence of a variable with a lower ranking is always left of a variable of higher ranking, and the pattern is in *canonical form*. The pattern λ is not in *canonical form*, if the variables are ordered by their letter in the alphabet. By renaming the variables B to A, C to B, A to C and D stays D, the pattern μ in *canonical form* is built in the example below.

$$\lambda = aBaCAaDaB$$
$$\mu = aAaBCaDaA$$

2.4.3 Pattern Matching

Both algorithms for the *matching problem* require to find the occurrence of a word inside another word. This problem is generally called *pattern matching* and it can be differentiated between finding the first occurrence of a specific word or all occurrences, depending on that are different algorithms possible as a solution.
The Knuth Morris Pratt algorithm defined by Knuth et al. [7] and the Aho-Corasick algorithm by Aho et al. [8] both have a linear run time with respect to the length of the two words, while the Rabin-Karp algorithm by Karp and Rabin [9] varies, since it uses hash functions, so its worst case is up to the product of the length of the input words.

Chapter 3

Methodology

3.1 Algorithm for Matching Problem for Regular Patterns

This section explains how the algorithm 1 for deciding whether a pattern matches a word works. It was defined by Fernau et al. [4], and is implemented in the next chapter.
The main idea of the algorithm is to find all terminals from the pattern in the word. The other parts of the word will be filled by variables from the pattern, so they are not preventing the possible match. The algorithm traverses the pattern from left to right, and whenever a terminal is found it needs to also be found in the word, else the pattern and the word do not match.

Algorithm 1 Solving the matching Problem for Regular Patterns

```
Input: regular pattern $\alpha$, and a word $w$ with $n = |w|$
Output: $true$ or $false$, depending whether $\alpha$ matches $w$ or not
 1: split $\alpha$ into $\alpha = w_0 \prod_{\text{i=0, m-1}}(X_i w_i)$ with $m \geq 1$, $X_i \in var(\alpha)$ and $w_i \in \Sigma^*$
 2: $s := 1, j := |w_0|$
 3: if $w_0$ != $w[0..j]$ then return false
 4: while s < m do
 5:     if $w_s$ = ε then
 6:         $j := j + 1$
 7:     else
 8:         if $w_s$ is not in $w[j+1..n]$ then return false
 9:         let $j\prime$ be the start position of the leftmost occurrence of $w_s$ in $w[(j+1)..n]$
10:         $j := j\prime + |w_s| + 1$
11:         $s := s + 1$
12:     end if
13: end while
14: if $w_{\text{m-1}}$ != $w[(n - |w_{\text{m-1}}|)..n]$ then return false, else return true
```

To achieve that, is the pattern factorized into its variables X_s and the terminals between the variables w_s. The first one w_0 and the last one $w_{\text{m-1}}$ are especially important, as they are the prefix and the suffix of α, and if α matches w, are they also a prefix and suffix of w. This property is checked in line 3 and 14 respectively.
j points to the first terminal of the word that was not already matched by the prefix, and it represents the portion of the word that can still be used to match terminals from the pattern. Then the algorithm iterates through the remanding w_s of α, while minimizing the needed terminals to match the w_s.

If any w_s is empty, it means that there is no fixed terminals that need to be in the found word, but since X_s must be replaced by a non-empty word, advances j by one terminal from the word. If w_s contains one or more terminals, the position of this is w_s in the remanding word searched, the remaining word is indicated by the position of j. This is done with one of the pattern matching algorithms from section 2.4.3. If the w_s is found, it increases j up to the point where the first occurrence of w_s ended. j even gets increased by one more, since after w_s is another variable in the pattern, which gets filled by one terminal from the word.

This is done for all w_s, if every w_s is found after each other, then $w \in L(\alpha)$, else it they do not match.

3.2 Matching Problem for One Repeated Patterns

The second algorithm that is going to be implemented is for the matching problem of one repeated patterns, it is explained in the following section.
The original algorithm to solve the matching problem for one repeated patterns by Fernau et al. [4] is defined with two different versions, one in cubic run-time, and an extension of that in quadratic run-time. The problem of the slower version is that all *factors* of the word get iterated, whereas the faster version provides an additional algorithm to skip some of the factors. Only the slower version will be considered here.
The idea of this algorithm can be compared to the algorithm for regular patterns, when a good replacement for the repeating variable is found, the pattern can be traversed similar to how in the matching problem for regular patterns. Therefore, is the pattern split into parts that contain the repeating variable, parts with non-repeating variables and parts of only terminals. This is called *factorization* and the formal definition is shown in 3.1.

$$\alpha = w_0 \prod_{\text{i=1,m}} (\beta_i \; w_i \; \gamma_i \; w_i') \beta_{\text{m+1}} w_{\text{m+1}} \tag{3.1}$$

m depends on the number of blocks that are needed to perform the factorization, the rules for the partitions are, that w_0, w_i and $w_m + 1$ only contain terminals if they are not empty. Furthermore, β_j contains only non-repeating variables and terminals, while γ_j contains only the repeating variables and terminals. Also begin and end β_j and γ_j always with their respective type of variable, this is also the case when they only contain a single variable.
This factorization is needed in the preprocessing phase necessary to find right placement of the b_j while the pattern is traversed. This is done with the matrix M, its dimensions are the length of the word and the number of b_j that are needed for the factorization, which results in a $|w| = n$ x $m + 1$ matrix since there is a total of $m + 1$ b_j's. $M[i][j]$ describes the shortest word from the position $i \in n$ that can be matched by $b_j 1 \leq j \leq m + 1$.
This M is used later in the algorithm to see what part of the word is needed for b_j, so it is clear which parts of the word can be used to match the $w_i \; \gamma_i \; w_i'$, and which do not contain any other variable then the repeating one. When the repeating variables is replaced by a factor of the word can a regular pattern matching algorithm be used, and it works similar to the algorithm for regular patterns.

The pseudo code for the preprocessing and the algorithm, as well as the formal proof, can be found in the paper by Fernau et al. [4].

3.3 Descriptive Pattern

The final algorithm that is going to be implemented, is the one for finding a descriptive pattern of a sample under a given pattern class.
Given a pattern class Π and a sample of words S, the Π-*Desc-Pat* algorithm learns a descriptive pattern. Therefore, it starts with the least descriptive pattern, that still matches all words from the sample. This pattern is a list of n distinct variables, where n is the length of the shortest word S, if the pattern had less variables, it would not match all words from the sample. So if the sample had nothing in common and followed no pattern, would this pattern be the descriptive one.
Otherwise is the pattern not descriptive and can be improved by two options, first by replacing a variable with a terminal from the shortest word, then by replacing a variable X_i with a variable X_j that occurred before X_i. If either of those steps leads to a pattern that is still in the pattern class Π and also matches all the words from the sample, then it is better than the old version without the replacement. This is true, because a variable gets deleted in both cases, which means the pattern language gets smaller, and only the words with exactly this replacement function are still in the pattern language, while words that need another replacement are no longer in the pattern language.
After all replacements are done, the is descriptive patter found [5].

3.4 Data sets

There are two different data sets that will be used in the evaluation, on the one hand random data, and on the other hand nucleic acid sequences. The random data is generated by a pseudorandom number generator, it is possible to produce regular and one repeated patterns, the parameters for the generation include the length of the pattern, the number of different variables in it and the alphabet from which the terminals are generated. Furthermore, it is also possible to generate a word from a given pattern, where the range from which the number of terminals that are used to replace a single variable are set.

The nucleic acid sequences are used to analyse the relationship between different related subspecies, they taken from a database by the university of Wuerzburg [10], where sequences for many different subspecies are stored. A nucleic acid sequences is a blueprint to build a protein, each sequence contains a sequence of *bases*. There is a total of four bases, and the function of the final sequence is only defined by the order of these bases. The sequence for one protein is then downloaded from the database for multiple subspecies, and can then be used to learn the pattern for that one class of species. The learned pattern can then be compared to the sequence of another subspecies of the same class.

3.5 Measurement techniques

There are two main questions that need to be addressed to answer the research questions, the first being whether a what information can be extracted from a learned Π-*Desc-Pat* pattern. This can be described by defining a metric for patterns, depending on their amount of information.
The easiest way to measure a pattern is to count the numbers of variables and terminals in respect to the length of the pattern. This works similar to like the idea of the algorithm to find a Π-*Desc-Pat*, the less variables are in the pattern, the smaller is the pattern language, which means that the pattern is more specific. This results in a higher metric score, because the pattern is rated by how many words that do not belong to the sample are still matched. This metric will be called *varCount*.
Π-*Desc-Pat* always results a descriptive pattern, so a pattern for a given sample is always ideal to this condition, but this metric is used to compare patterns from different sample types, or the two pattern classes against each other.
Important is that Π-*Desc-Pat* always results a descriptive pattern, so a pattern for a given sample is always ideal to this condition, but this metric is used to compare patterns from different sample types, or the two pattern classes against each other.

The second measurement problem considers how close the results of the Π-*Desc-Pat* algorithm is to an previous defined pattern. To test that is an original pattern used to create words and then are the words used as a sample to generated a Π descriptive pattern. Afterwords can be counted how many words generated from the original pattern are matched by the new pattern, this will be called *learn and check*. The number of words used to learn the new pattern can be varied, furthermore can the way how the words are generated from the sample be changed.
Another way to compare the original and the generated pattern is the *longest common substring* metric, this is defined as the longest continues segment in both patterns.

Chapter 4

Implementation

4.1 Used Tools

The first decision for the implementation was the question of programming language, Python [11] was chosen. The main reason for that was an already existing Python implementation for the pattern matching algorithms from section 2.4.3, on which could be build upon. It was done in a previous project, and together with the existing experience in Python, made Python a good choice. But there was no reason found to use another programming language, since the goal was to see how the algorithms perform in practise and what the learned pattern looks like for different input data. Therefore, was raw performance not that important, and it was not necessary to consider switching to a lower level programming language like C.
Overall makes the good usability of Python a good choice for a proof of concept project like this, but there was nothing that would prevent another programming language.

The implementation was written with Visual Studio Code [12] and executed on a Windows 10 machine. This should not impact the execution on another setup, but file paths are defined differently for Windows and Linux machines in Python, so problems might occur when the test cases that output to a file are used in Linux. Python version 3.9.6 was used, but any version above 3.8 is expected to work, an older version is not recommended since the self-documenting feature for f-string from version 3.8 is used.
For the projects itself are no external libraries needed, but the packet "matplotlib" is required to build the graphs to show of the results. Also the packages "timeit" and "wget" are used in some parts of the old project for pattern matching, but they are not needed for the implementation of this work. The source control tool "git" was used through the Git integra-

tion for Visual Studio Code, the final repository can be viewed at GitHub under the address `https://github.com/MareikeBurg/LearningPatternsfromTextualData`, or in the attached Zip file.

4.2 Structure

The implementation is separated into different python files depending on their use case. The first one is the *learning_Util.py* file, it contains mostly operations on a single pattern or word and provides these methods as utility for other functions. It contains the function to find the maximal terminal factors of a pattern for example, and the functions to switch between the two types of representations for pattern that are defined later.
The next file is the *learning_Patterns.py* file, it has the implementations of the functions for the algorithms defined in 3. The first two files together can be used on their own to apply all three algorithms to a project of choice, their only dependency is the existence of an implemented *pattern matching* algorithm.
Next are the files *learning_randomdata.py* and *learning_biodata.py*, they are used to build data sets on which the algorithms are tested. Testing is done in the *pattern_testing.py* file, where multiple test cases are located, it also contains the metric functions.
The other files all belong to the old pattern matching project, *pm_aho_corasick.py*, *pm_boyer_moore.py*, *pm_knuth_morris_pratt.py* and *pm_rabin_karp.py* all contain a complete implementation for the pattern matching problem of finding a single word in a longer word, so they could be used in *learning_Patterns.py*.

4.3 Patterns

The first goal for the implementation is to store the symbols of a patterns. The easiest way would be an object of type String, it contains objects from type *character*, each lower case character would imply a *terminals* and each upper case character a *variables*. This enables a good way to differentiate between *terminals* and *variables*, since there are the native and fast Python functions *isUpper()* and *isLower()*. The string is also good readable and enables a very neat output. This representation is called *alphabet* form.
The problem is that this system is limited to 26 *terminals* and *variables* each, therefore only small *alphabet* sizes would be possible. Furthermore, the algorithm to solve Π-*Desc-Pat* begins by a pattern that is consisting of different variables, and its length is the length of the shortest word. Therefore could the shortest word not be longer then 26 symbols with the alphabet system.

This is solved with the *integer list* form, as the name suggest are the symbols stored as integers in a Python list. The datatype character is nothing else then an *Integer* in the system memory, so it is possible to expand the alphabet by adding more possible integers. So like in the *ASCII* system, it is possible to define a range from x to y as variables and another range from y+1 to z as literals. But this leads to a fixed limit at the start of the program, so it is better to not separate them by a given offset and instead split between even and uneven numbers.
This also enables an efficient checking function, by using $modulus 2$. Another possibility would be to split between positive and negative numbers, but this was not done because the output with the negative signs does not look good.
In the implementation is only the integer list format used, but there are two functions to convert between the formats, with an example on how to use them shown in 4.1. For the symbols where no character in the alphabet exists, is the letter Z/z with its integer value behind it used, so that all symbols can be converted.

```
import learning_Util.py
>>> convertToIntList("aAa")
[1, 2, 1]
>>> convertToAlphabet([2,4,2])
'BCB'
>>> convertToAlphabet([48, 49, 100, 1001])
'YyZ100z1001'
```

Listing 4.1: Representations for patterns

4.4 Membership of patterns in a pattern class

Before the algorithm for Π-*Desc-Pat* can be implemented, is an algorithm for the *membership problem* needed. This is done similarly for both *regular* and *one repeated patterns*, in both cases iterates an algorithm over the pattern and counts how many times each variable occurs, which is done by storing the variables that were already found in a list. If any variable is found twice for a regular pattern, then is the pattern not regular and the algorithm returns. For one repeated patterns is the first variable that occurs twice stored separately and only if another variables occurs twice the answer is false.

If the whole pattern can be traversed, then is the pattern in its pattern class and true is returned. The implementation for one repeated variables is shown in 4.2, aswell as two examples with a one repeated pattern and a not one repeated pattern.

```
def isOneRepPatternClass(pattern):
    repeatedVar = None
    marked = []
    for c in pattern:
        if isVariable(c):
            if c in marked:

                if repeatedVar is None:
                    repeatedVar = c
                elif c != repeatedVar:
                    return False
            else:
                marked.append(c)
    return True
>>> isOneRepPatternClass([0,1,2,0])
True
>>> isOneRepPatternClass([2,2,4,4])
False
```

Listing 4.2: Checking if a pattern is one Repeated

4.5 Main Algorithms

4.5.1 Matching regular patterns

With the implementation of the pattern matching algorithms already done from the previous project, is the implementation of the matching problem for regular patterns possible, and it can be called via the *matchingRegular(pattern, word)* function from the $learning_patterns.py$ file. Importantly is to note that a variable can not be replaced by the empty word, which is why the second function call in example 4.3 correctly returns false, since the *terminal* 3 can not be matched with the word, because $w[1] = 3$ is already needed for the replacement of the *variable* 2 from the pattern.

```
>>> matchingRegular([1,2,3,4],[1,1,3,5,5])
True
>>> matchingRegular([1,2,3,4],[1,3,1,5,5])
False
```

Listing 4.3: Using the matchingRegular function

4.5.2 Matching one repeated patterns

The function call in the implementation for the matching problem of one repeated patterns can be called analogue to 4.3, but for it to work, are three functions necessary. The first one is the preprocessing function, it generated the matrix M according to its definition in 3.2. The convention was made that when the value of $M[i][j]$ can not be defined since the b_j does not appear after a certain i, the value *None* is stored. So inside the main loop of the algorithm has to be checked if M is *None* or not.

The next helping function is needed to create the *factorization* of the pattern α as defined in 3.1, the factorization gets stored into a *dictionary*. So the different parts like all γ_j or the $\beta_{\mathrm{m+1}}$ are accessed by a predefined key. This is done to have an easy way to access all the different parts and always have a good way to debug.
How this looks like in the end result is seen in 4.4, it is for the pattern $[1,2,1,2,1]$, with the repeating variable 2, and the word $[1,5,1,5,1]$.

```
M = [[None, None], [0, 0], [1, 1], [2, 2], [3, 3]]
factorisePattern(pattern, repeatingVar) = {'w0': [1], 'betaList': [[]], 'wiList
    ': [[]], 'gammaList': [[2, 1, 2]], 'wiDashList': [[1]], 'betam+1': [], 'wim
    +1': []}
```

Listing 4.4: Preprocessing for matching of one repeated patterns example

The purpose of the final function is to find the repeating variable, which is necessary to create the *factorization*. This can be done very similar to the function *isOneRepPatternClass* from section 4.4, that traverses the pattern and stores the first variable that occurs twice, so the new function *findRepeatedVar* can return the variable at that point.

By using this three helping functions from above, can the main algorithms be completed aswell, the final function is called by *matchingOneRep(pattern, word)*.

4.5.3 Learning Π descriptive pattern

Since everything that is necessary for the algorithm to learn Π descriptive pattern is established, can the algorithm be implemented straight forward. The input for the algorithm is a sample to learn from, which is a list of words, and two functions specific to the pattern class Π. They are needed to verify if any of the improvement steps was successful or not, therefore a function for the matching problem Π, and a function for the membership problem of Π must be used.
But they are already implemented, so 4.5 shows how a pattern is learned, the functions names for the matching and membership problems are given as a parameter, if none are given it is defaults to the functions for the regular pattern class.

```
>>> sample = [[1,3,5],[1,3,3,5],[1,3,3,3,5]]
>>> descPat(sample, matchingFunction=matchingRegular, classMembershipFunction=
    isRegularPatternClass)
[1, 3, 0]
```

Listing 4.5: Calling the function to learn a Π descriptive pattern from sample

Chapter 5

Evaluation

5.1 Viability test

The first test is used to get an overview what is possible with the implementation and what the limiting factors are. Some arbitrary values are set as the baseline and then is checked what happens when one of the values is changed.
This is shown for the matching problem table in 5.1, the raw data is found in the */src/data/-manualResults* folder. The test works by generating a random pattern and tries to match it with a random generated word, the time it takes to complete the matching algorithm is saved. This is repeated for multiple words. There are two options to generate the words, the first one is called *random* and it generates a completely random word, therefore will the algorithm most likely return false. For *matching* words is the pattern taken and a word is build from the pattern, so the pattern always matches the word. By this it can be tested how long it takes for the pattern to deny not matching words, and how long it takes to accept words.
This process is repeated for multiple patterns and the average is taken.

	baseline	increased pattern length	increased number of variables	decreased alphabet size
Regular				
random	0.00050	0.03380	0.00082	0.00061
matching	0.00125	0.09563	0.00153	0.00131
One repeated				
random	0.33598	-	0.32430	0.21333
matching	0.62453	-	0.60169	0.48026

Table 5.1: Results of viability test

As it can be seen in the table 5.1, are the regular patterns way faster then the one repeated ones. This was expected, since the algorithm for regular patterns had a linear runtime and the one for one repeated patterns only a cubic one. Furthermore, it takes way longer to verify a matching pattern, which can be explained by the fact that most of the random patterns can be sorted out after only a few terminals.

5.2 One Repeated Matching Naive

Because there were continues problems with the algorithm to solve the *matching problem* for one repeated patterns, a brute force algorithm for the algorithm was designed. It is called *matchingOneRepNaive* and its idea was to set the repeating variable to a fixed factor of the input word, and then treat the pattern as a regular pattern. If one of the resulting patterns matched the word then the pattern is one repeated, and the replacement function for the repeating variable to construct the word is the factor. If no such factor exists, then does such a function not exit, and they do not match.
As expected was the runtime of this algorithm not practical though, but it proved that the past algorithm for one repeated pattern sometimes had wrong results. On the other hand was it still slower then the past algorithm, so the possibilities for further testing were limited.

5.3 Measuring the generated patterns

The goal of the next test case is to see how useful a learned pattern from random data is. Therefore is a random pattern generated and a sample of words built by placing random

words into the pattern. This sample is used to learn a Π descriptive pattern and the original and the new patterns are compared.
In 5.1 is the first list the original pattern, the second list is the pattern that is generated from the sample of the original pattern. The third line contains the number of variables in the patterns, first the number in the original, then the number in the learned pattern. Since the random pattern was set to 5 variables does this not change regular patterns, for the one repeating variable does it change since the number of variables changes depending on the position of the repeating variable.

```
Regular patterns:
[0, 13, 3, 45, 29, 27, 19, 31, 2, 25, 3, 17, 25, 4, 19, 21, 39, 6, 8, 3] [0,
    13, 3, 45, 29, 27, 19, 31, 2, 25, 3, 17, 25, 4, 19, 21, 39, 6, 8, 3, 10, 3]
5 6
[35, 3, 15, 0, 35, 2, 37, 27, 4, 47, 6, 39, 41, 39, 27, 45, 39, 21, 8, 27] [35,
     3, 15, 0, 35, 2, 4, 6, 8, 27, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30,
    27]
5 16
One repeated pattern:
[0, 5, 47, 0, 9, 25, 4, 15, 1, 6, 15, 8, 27, 10, 11, 13, 3, 15, 3, 33, 17, 12,
    0, 0, 14, 0, 0, 16, 18, 15] [0, 5, 47, 2, 9, 25, 4, 5, 6, 15, 8, 10, 12,
    14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50,
     52, 54, 56, 58, 60, 15]
14 31
```

Listing 5.1: Comparing the original and a learned pattern

The complete testset is found in the files *varCountTestRegular* and *varCountTestOneRepeated* which are located in the */src/data/manualResults* folder. The result over all patterns averaged a number of 14.2 variables for learned regular patterns, and 31.2 variables for learned one repeated patterns.

One possible explanation for this is in the structure of the generated one repeated patterns, none of them had their repeating variable at the same position as in their respective original pattern. Instead were large parts of the new pattern only one variable after another variable, this also happened for the regular patterns but not that strong.
The lack of a repeating variable indicates that the implementation is not working as it should.

But for both types of pattern is visible that the majority of terminals is at the beginning of the new pattern, which makes sense in the context how the algorithm works, since it starts replacing at the beginning.

5.4 Learn and match

This testing method is based on the second measurement method from chapter 3.5, it tests how good a sample generated by a pattern, can be used to learn the original pattern. This test was done with using between 1 and 60 words to learn the new pattern, the result for each number of variables is shown in the graphic 5.1.
As expected can not many words be matched when the sample size for Desc-Pat is very small, but surprisingly happens the same after to many words aswell. This means that there is a sweat spot where the most words are matched by the new pattern, for this exact numbers is it around 10 that are used as sample.

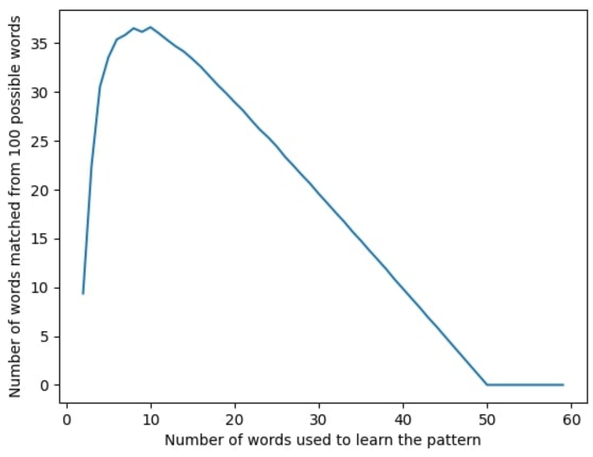

Figure 5.1: Matching words from the original pattern by learned pattern

5.5 Using patterns to find common descent

As described in section 3.4 are also nucleic acid sequences for a protein available, they were used for four different types of worm, the results were comparable for all four. The original data is located in the *bioSamples* folder, the extracted sequences and the learned pattern is saved in the *manualResults* folder.

5.2 shows the patterns that were learned from the sample, for better readability is the *alphabet* form used. It can be seen that the learned patterns disparate in number of variables and therefore in their precision, the pattern of the *Holtermaniella* contains long continues sequences of terminals, which shows that the protein structure only differentiate on single positions. This makes the pattern pretty useful and it could be used to test other subspecies with it.

```
Tremella:
AtggBctCcgDccEcFaGgHaI

Holtermaniella:
gtgaacaacctcaaccttgattggttttacAatcaaggcttggaBttgggtgttgctgctgcaaagtggctcgcctt
aaaaCcatDagEtggFcttgGctatgaHttIgtJgKtLgMgN

Allocreadiata:
AgcttaBaaaCtatDcEcFtGgH

Filobasidium:
aAataBcaCccaDgacE
```

Listing 5.2: Learned pattern for different classes of worms

The results for the other classes of species seem to be not that valuable, the information that a sequence of tgg for example, occurs somewhere in the beginning of the pattern, is not that useful. Alternatively could the subspecies also not be related that closely, which could explain why the sequences are more different, but the input data does not look like it.

The first 30 characters of *Allocreadiata* are shown in 5.3 for three picked subspecies. The only difference between them is one single base at two different positions, but since there are over 85 subspecies, add these differences up and the learned pattern contains less and less variables. A more flexible model might be needed to further refine this model.

```
ggcttataaactatcacgacgcccaacaag
                          ^
ggcttataaactatcacgacgcccaaaaag
                        ^
ggcttataaactatcacgacgccctacaag
```

Listing 5.3: Excerpt of sequence of subspecies from Allocreadiata

Chapter 6

Conclusion

6.1 Problems

One of the biggest problems was to validate whether the algorithms work correctly, especially for the matching problem of one repeated patterns. The factorization has so many corner cases that it is really hard to find all of them. Therefore, it might look like that the algorithm is working, but for some cases it does not, and patterns get not accepted as one repeated, while they in fact are. This could explain, why the learned one repeated patterns often had no repeating variable.
It could have been solved by building a large testing class, where all problematic corner cases are noted. My used method was to check a couple cases at one time, so it was possible that after fixing one case, another case would break again that worked before. This proved to be insufficient for the complexity of the problem.

Aside from that were often many variables directly after each other in a learned descriptive pattern, which can be seen in 5.1. This might not prevent the pattern from being descriptive, but is not ideal and could interfere with metrics to measure the learned pattern.

6.2 Final words and future work

The algorithm of learning descriptive patterns for regular patterns seems to work. It is feasible and efficient to construct such a pattern from a sample, while the learning of one repeated pattern worked partially.

The slow version of the algorithm for matching one repeated pattern proved to be not that practical, since it was to slow for many test cases and prohibited a good comparison. Maybe this could be solved by the full version of the algorithm, but that would also add another layer of complexity. This hints that research should focus on other pattern classes.

Another topic is the possible combination for input data, there are countless possibilities for data sets and there are many properties that could be altered. This includes the length of a text, the alphabet of it and also the possibility of each symbol. This thesis tested some of the properties, like the length, but a testing system for more of them is still to be found.

The learn and match test procedure revealed an interesting property of learning a pattern from a sample, since a bigger sample seams not always ideal to reassemble the original pattern.

The method of studying relationship patterns from the genetic code showed that one of the class of species had a closer ancestry, so the procedure is proven to be viable. It would be useful if a model exists that enables the patterns to be a bit more flexible, for example with a model that assigns a score based on how many terminals are matched. This score would still be high for very related subspecies, and could handle the problem of having one different base in an otherwise completely equal sequence.
Counting the variables in a pattern was a successful method to measure the quality of a learned pattern in textual data, but further approaches are possible.

Bibliography

[1] S. C. Kleene, “Representation of events in nerve nets and finite automata.” [Online]. Available: https://www.rand.org/pubs/research_memoranda/RM704.html

[2] C. Câmpeanu, K. Salomaa, and S. Yu, “A formal study of practical regular expressions,” vol. 14, no. 6, pp. 1007–1018, publisher: World Scientific Publishing Co. [Online]. Available: https://www.worldscientific.com/doi/abs/10.1142/S012905410300214X

[3] D. D. Freydenberger, “Extended regular expressions: Succinctness and decidability,” vol. 53, no. 2, pp. 159–193. [Online]. Available: http://link.springer.com/10.1007/s00224-012-9389-0

[4] H. Fernau, F. Manea, R. Mercas, and M. L. Schmid, “Pattern matching with variables: Efficient algorithms and complexity results,” vol. 1, no. 1, p. 38.

[5] H. Fernau, F. Manea, R. Mercaş, and M. L. Schmid, “Revisiting shinohara’s algorithm for computing descriptive patterns,” vol. 733, pp. 44–54. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0304397518302731

[6] D. Angluin, “Finding patterns common to a set of strings,” vol. 21, no. 1, pp. 46–62. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/0022000080900410

[7] D. E. Knuth, J. H. Morris, Jr., and V. R. Pratt, “Fast pattern matching in strings,” vol. 6, no. 2, pp. 323–350. [Online]. Available: http://epubs.siam.org/doi/10.1137/0206024

[8] A. V. Aho and M. J. Corasick, “Efficient string matching: an aid to bibliographic search,” vol. 18, no. 6, pp. 333–340. [Online]. Available: https://dl.acm.org/doi/10.1145/360825.360855

[9] R. M. Karp and M. O. Rabin, “Efficient randomized pattern-matching algorithms,” vol. 31, pp. 249–260. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.86.9502&rep=rep1&type=pdf

[10] “Database for ribosomale rna,” http://its2.bioapps.biozentrum.uni-wuerzburg.de/, last access on 2021-09-01.

[11] “Python,” https://www.python.org/, last access on 2021-09-01.

[12] “Virtual studio code,” https://code.visualstudio.com/, last access on 2021-09-01.