BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

RECOGNITION OF REPEATING SMS PATTERNS

ROZPOZNÁVÁNÍ OPAKUJÍCÍCH SE VZORŮ SMS ZPRÁV

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR JAKUB KOČALKA
AUTOR PRÁCE

SUPERVISOR Mgr. LUKÁŠ HOLÍK, Ph.D.
VEDOUCÍ PRÁCE

BRNO 2021

Brno University of Technology
Faculty of Information Technology

Department of Intelligent Systems (DITS) Academic year 2020/2021

Bachelor's Thesis Specification

24169

Student:	**Kočalka Jakub**
Programme:	Information Technology
Title:	**Recegnition of Repeating SMS Patterns**
Category:	Algorithms and Data Structures

Assignment:

SMS spam campaigns are generated by instantiation of simple patterns, essentially simple regular expressions. The goal of this work is to propose a method for improving the precision of a method for classification of messages to campaigns used in the Mavenir company.

1. Familiarise yourself with the classification methods used in Mavenir for detection and classification of SMS spam campaigns and with the Smith Watermann algorithm [1].
2. Implement a version of Smith-Watermann algorithm, adapted and optimized for inferring the SMS-campaign patterns.
3. Evaluate efficiency of the algorithm and its impact on the precision of the classification method of Mavenir, discuss the potential of your solution for practice.

Recommended literature:

1. Smith, Temple F. & Waterman, Michael S. "Identification of Common Molecular Subsequences". Journal of Molecular Biology. 147 (1): pages 195-197. 1981. doi:10.1016/0022-2836(81)90087-5.

Requirements for the first semester:

1. Item 1 of the assignment,
2. initialisation of item 2,
3. at least a part of the text of the thesis.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Holík Lukáš, Mgr., Ph.D.**
Head of Department:	Hanáček Petr, doc. Dr. Ing.
Beginning of work:	November 1, 2020
Submission deadline:	July 30, 2021
Approval date:	November 11, 2020

Abstract

With the advances in e-mail spam recognition and user awareness, spammers are moving towards less researched media. One of those is the short messaging system (SMS), which boasts high availability and open rates. Those characteristics are also attractive to legitimate businesses that need to send short, bulk messages to their clients. However, while these messages might be solicited by the end-user, they might represent a loss for the SMS service provider, as these businesses often misuse unlimited SMS plans meant for regular customers to avoid paying for more expensive solutions designated for them. It is therefore desirable to be able to recognize both unsolicited and solicited bulk messages. Bulk messages are generally generated from a template. The goal of this work is to design a clustering algorithm that treats a message as a sequence of lexical units (words), and evaluate it's effectiveness compared to a locality sensitivity hashing method that treats the message as a string of symbols. The work evaluates the suitability of the Smith-Waterman alignment algorithm for this task. The work details why Smith-Waterman (and other local alignment techniques) is unsuitable, and how it can be replaced by Needleman-Wunsch (global alignment) to produce much better results. The resulting algorithm is able to cluster real messages into campaigns satisfactorily, and performs well even in situations where the benchmark locality sensitivity hashing method fragments campaigns.

Abstrakt

Vďaka pokroku v rozpoznávaní spamu v e-mailoch a zvyšovaní povedomia používateľov smerujú spameri k menej preskúmaným médiám. Jedným z nich je *short messaging service* (SMS). Táto služba poskytuje užívateľom možnosť reagovať na správy v krátkom čase a v skoro ľubovolnom prostredí. Tieto vlastnosti sú atraktívne aj pre legitímne podniky, ktoré potrebujú svojim klinetom zasielať krátke hromadné správy. Aj keď sú tieto správy z pohladu koncového užívateľa vyžiadané, pre poskytovateľa služieb SMS môžu predstavovať stratu, pretože tieto podniky často zneužívajú neobmedzené SMS plány určené pre bežných zákazníkov, aby sa vyhli plateniu za pre nich určené, ale drahšie produkty. Je preto žiaduce vedieť rozpoznať nevyžiadané aj vyžiadané hromadné správy. Hromadné správy sa zvyčajne generujú zo šablóny. Cieľom tejto práce je navrhnúť zhlukovací algoritmus ktorý správy analyzuje ako sekvencie lexikálnych jednotiek (slov), a vyhodnotiť jeho efektivitu v porovnaní s *locality sensitivity hashing* metódou ktorá správy analyzuje ako reťazce symbolov. Práca vyhodnocuje vhodnosť algoritmu Smith-Waterman pre túto úlohu. Práca popisuje, prečo je Smith-Waterman (a ďalšie lokálne zarovnávania) nevhodný, a ako je možné ho nahradiť algoritmom Needleman-Wunsch (globálnym zarovnávaním), aby sa dosiahli oveľa lepšie výsledky. Výsledný algoritmus dokáže uspokojivo zhlukovať skutočné správy do kampaní a funguje dobre aj v situáciách, kde *locality sensitivity hashing* kampane fragmentuje.

Keywords

Smith-Waterman, Needleman–Wunsch, SMS, spam, sequence alignment, string clustering

Kľúčové slová

Smith-Waterman, Needleman–Wunsch, SMS, spam, zarovnanie sekvencií, zhlukovanie reťazcov

Reference

KOČALKA, Jakub. *Recognition of Repeating SMS Patterns.* Brno, 2021. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Mgr. Lukáš Holík, Ph.D.

Rozšírený abstrakt

Rozpoznávanie nevyžiadaných hromadných správ (spamu) v emailovej komunikácii je v dnešnej dobe veľmi účinné. Napomáha tomu aj fakt, že užívatelia sú o nebezpečenstvách email spamu dobre poučený. Spameri preto prechádzajú na iné médiá. Jedným z nich je SMS (short messaging service). Na rozdiel od telefonickej a emailovej komunikácie, užívatelia sú schopný interagovať s SMS v skoro ľubovolnom prostedí, a nevyrušovať pritom ostatných. Interakcia s SMS správami je taktiež skoro instantná: 90% správ je otvorených, a väčšina je otvorená do 15 minút. Rozpoznávanie spamu v SMS správach je menej preskúmané ako v emailoch. Užívatelia sú tiež menej poučený o existencii a povahe spamu v SMS. Užívatelia tiež považujú SMS za dôverihodnú službu, čo zvyšuje mieru odozvy na SMS spam.

SMS ako médium pre spam nabralo na popularite so zvýšenou dostupnosťou neobmedzených predplatných SMS plánov. Toto predstavuje ďalší problém: spoločnosti ktoré potrebujú klientom posielať krátke hromadné správy sa obracajú na SMS. Veľa z nich využíva neobmedzené plány určené pre bežných užívateľov, namiesto pre nich určených drahších produktov. Týmto vzniká poskytovatelom škoda, ako ušlím ziskom, tak zvýšenou záťažou na sieť. Je teda žiadúce vedieť odhlaliť nevyžiadnané aj vyžiadané hromadné správy.

Hromadné správy sú posielané vrámci takzvaných kampaní. Správy v jednej kampani sú väčšinou vygenerované zo šablóny s fixními aj variabilnímy prvkami. Ak sú vygenerované správy syntakticky veľmi podobné, môžme ich zhlukovať do kampaní pomocou podobnosti textu. Pre úspech kampane je ale skôr dôležitá sémantická podobnosť správ. Útočníci cielene vkladajú do kampaní variabilitu v podobe synoným a šumu. Toto spôsobuje problém v zhlukovacích metódach zvaný fragmentácia: vytvorenie viacerých zhlukov z vzoriek patriacich do rovnakého zhluku.

Jednou z metód na zhlukovanie textu je takzvaný *locality sensitivity hashing* (LSH) založený na podobnosti textu. Táto metóda je rýchla, ale je náchylná na fragmentovanie variablných kampaní. Cielom tejto práce je vytvoriť zhlukovaciu metódu založenú na spracovaní správy po slovách, nie po písmenách. Navrhnutý algoritmus je založený na zarovnávaní reťazcov, metóde využívanej v bioinformatike na vyhladávanie podobných sekcií v proteínových reťazcoch. Navrhnutý algoritmus je schopný zhlukovať správy do kampaní, a zároveň sa iteratívne naučiť šablónu kampane. Algoritmus sa učí z pozitívnych vzorkov, ale je súčastne schopný odhaliť negatívne vzorky, a zamedziť ich ovplineniu šablóny (nie je ale schopný zdokonaliť pomocou negatívnych vzorkov šablónu).

Vrámci tejto práce bola preskúmaná vhodnosť zarovnávacieho algoritmu Smith-Waterman na učenie sa šablón. Ukázalo sa však, že tento algoritmus je nevhodný, pretože vykonáva lokálne zarovnanie. Ďalej bol odtestovaný globálny zarovnávací algoritmus Needleman-Wunsch. Tento prístup bol úspešný, schopný vytvoriť akceptovatelné zhluky a použitelné šablóny. Výsledný algoritmus bol úspešný v zlúčení kampaní ktoré LSH metóda fragmentovala.

Recognition of Repeating SMS Patterns

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Mr. Lukáš Holík. Suplementary information was provided by Mr. Peter Salomoun and Ms. Hana Šimková. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
Jakub Kočalka
July 19, 2021

Acknowledgements

I would like to thank Mr. Lukáš Holík for his supervision of my work, as well as Mr. Peter Salomoun and Ms. Hana Pluháčková for provided advice.

Contents

1 Introduction **2**

2 Strings, Languages, and Tokens **4**
2.1 Language 4
2.2 Tuples 5
2.3 Tokens and Token Alphabets 5
2.4 Edit Distance 6

3 Alignment **8**
3.1 Spaces and Gaps 8
3.2 Needleman-Wunsch 9
3.3 Smith-Waterman 11
3.4 Parameters of Alignment Algorithms 11

4 SMS and SMS spam **14**
4.1 SMS 14
4.2 Spamming 14
4.3 SMS Spam 15
4.4 Spam Campaigns 15
4.5 SMS Spam detection in Mavenir s.r.o. 16

5 Algortihm for Inferring Sequence Generation Patterns **19**
5.1 Classification Algorithms 19
5.2 Tokenization 21
5.3 Campaigns 23
5.4 Scoring 23
5.5 Alignment Collapsing 26
5.6 Alignment Judging 26
5.7 Metrics 27

6 Optimizing the Algorithm **30**
6.1 Implementation 30
6.2 Naive Approach 31
6.3 Challenges 32
6.4 Hyperparameters of Second Degree 36
6.5 Improved Classification Algorithm 38
6.6 Generated Tempaltes 41
6.7 Merging Campaigns 42

7 Conclusion **43**

Chapter 1

Introduction

Sending unsolicited bulk messages - spamming - is an activity most associated with email communication. Spam protection in email is relatively well researched and effective. Users nowadays are also well educated on the dangers of email spam, and the response rate to email spam is dropping. This is driving traditional email spammers to seek out other avenues of distributing spam [10]. Due to the increased availability of unlimited plans, The Short Messaging Service (SMS) is one of them. While in some countries the use of SMS is on the decline, in others (Middle East, Africa, Asia) it is still popular.

SMS spam has received relatively small amount of attention from researchers in comparison to email. It presents it's own challenges. The common content based approach of spam filtering is ill suited for detecting SMS spam, as SMS messages are limited in length. Private companies now also send SMS to their customers much more, which can present a problem for mobile service operators as well, if they are not using the proper avenues. Some bulk messages might appear legitimate based on their content, but their detection is still of interest to providers, as their senders might be using cheaper plans designated for regular physical customers, therefore presenting both higher operating costs and loss in revenue.

Spam is usually sent in so-called campaigns. Messages within a campaign have the same purpose (e.g. advertisement), and also share a structure. This is because spam is usually generated from a template: a meta description of the message, with both fixed and variable fields. Each message is therefore personalized to the recipient, and a little different, but still sharing a similar overall structure. If we knew a template of a spam campaign, we could detect messages from that campaign easily by comparing them to the template. Conversely, by grouping messages together based on what template we think they were generated from, we can approximate the size of a campaign, and decide whether it is a spam campaign at all.

One approach to grouping messages into campaigns is to compute their text similarity. This is the approach that Mavenir s.r.o. currently uses. Apart from not learning a template, this approach is prone to error when a template generates messages with variable length or when noise is introduced to the messages. To combat this, an alternative approach based on analyzing the message as a sequence of words (as opposed to a sequence of letters) was proposed. The goal of this work is to implement an algorithm that can iteratively learn templates used to generate campaigns of messages and classify those messages into campaigns at the same time.

Chapter 2 defines some terms that will be useful in designing the algorithm.

To learn templates, we will attempt to locate similarities among sequences of words the messages contain - aligning them. Chapter 3 defines what is alignment, and describes two algorithms for aligning sequences, as well as a number of parameters of these algorithms and their meaning.

Chapter 4 explores the concepts of SMS and spamming in more detail. It also describes the method used for grouping SMS messages into campaigns in Mavenir.

Chapter 5 contains description of the proposed classification algorithm. Chapter 6 describes how the algorithm performs on real data, and how to adjust it to ensure it groups messages into campaigns correctly.

Chapter 2

Strings, Languages, and Tokens

In this section we will define some common notions and notations used in the field of formal languages. In Section 2.3 we build upon those to define new notions that will be more useful to the algorithms described later.

2.1 Language

This section has been adopted from [16]

An *alphabet* Σ is a finite, non-empty set of symbols called *letters*.

A *string* x over Σ is a finite sequence $x = a_1 \dots a_n$ of letters. Let $|x|$ denote the length of x, which is the number of letters of which the string consists (in this case $|x| = |a_1 \dots a_n| = n$).

The *empty string* is denoted ϵ. $|\epsilon| = 0$.

A *substring* of a given *string* is a contiguous string of *symbols* found in the *string*.

Certain string operations are defined:

- *Concatenation* of two strings x and y is $x \cdot y = xy$.
- *Power* $i \geq 0$ of string is defined recursively as follows:
  - $x^0 = \epsilon$
  - $x^i = xx^{i-1}$

A *language* is any set of strings. We define some language operations:

- *Concatenation* $L_1 \cdot L_2$ consists of all strings vw where $v \in L_1$ and $w \in L_2$
- *Kleene star* $L*$ consists of all words that are concatenations of zero or more strings in L

therefore a language is a subset of Σ^* (where $*$ is *Kleen Star operator*[11]).

2.1.1 Regular Languages

A *regular language* is a language over an alphabet Σ defined recursively as follows:

- the empty language $\emptyset$ and the language $\{\epsilon\}$ are regular.
- For each $a \in \Sigma$, the singleton language $\{a\}$ is regular.

- If A and B are regular languages, then $A \cup B$, $A \cdot B$, and A^* are regular languages.

A *regular expression* (or *regex*) over Σ denotes a regular language and is defined as follows:

- $\emptyset$ is a regex denoting the empty language.
- ϵ is a regex denoting $\{\epsilon\}$.
- a is a regex denoting $\{a\}$.
- Let r and s be regular expressions denoting languages L_r and L_s respectively. Then:
  - $(r.s)$ is a regex denoting $L = L_r \cdot L_s$.
  - $(r|s)$ is a regex denoting $L = L_r \cup L_s$.
  - $(r*)$ is a regex denoting $L = L_r^*$.

Note that a regular expression is a string over the alphabet $\Sigma_R = \Sigma \cup \{|\} \cup \{*\}$ (we simplified $(r.s)$ to just rs).

We say that a regular expression r matches a string s if and only if $s \in L_r$.

$$match(r, s) = s \in L_r$$

For the purposes of this paper we define several shorthands:

- $[s]$ matches any letter in string s
- . matches any letter in Σ
- $\backslash w$ matches any alphabetical character and an underscore
- $\backslash d$ matches any digit.
- $a? = \epsilon|a$
- $a+ = a(a*)$

2.2 Tuples

A *tuple* is a finite ordered sequence of elements. A special notation is used in this paper for defining tuples and referring to their elements. A (named) tuple is defined by specifying the names of each element. For example tuple t is a pair $(\mathit{elem1}, \mathit{elem2})$. To refer to elements of the tuple, we use a notation reminiscent of the dotted syntax for accessing attributes of objects in many object oriented languages. For example, to refer to the first element of pair t, we would write $t.\mathit{elem1}$.

2.3 Tokens and Token Alphabets

A *token category* $c \in \Sigma_T$ is a triple $(\mathit{pattern}, \mathit{priority})$ where $\mathit{pattern}$ is a regular expression describing a regular language over some alphabet Σ_0, $\mathit{priority} \in \mathbb{N}$. We say that c is a letter of Σ_T.

A *token alphabet* Σ_T is a directed rooted in-tree[1] of token categories.

Next we define

$$category(s, \Sigma_T) = \{c | c.priority = max(\{c'.priority | c' \in \Sigma_T \wedge s \in L_{c.pattern}\}) \wedge c \in \Sigma_T \wedge s \in L_{c.pattern}\}$$

In other words, $category(s, \Sigma_T)$ is a set of token categories with the highest priority among all categories in Σ_T whose patterns describe a language to which s belongs. Note that $category(s, \Sigma_T)$ will:

- be empty if there is no category in Σ_T that matches s
- have precisely one element if there is no category in Σ_T that matches s, and all categories of Σ_T with the same priority have patterns that describe languages that are mutually exclusive

A *token* t is a pair $(category, value, alt_count, optional)$, where $category$ is a token category, $value$ is a set of strings, $alt_count \in \mathbb{N} \cup \{-1\}$ is the alternative count, and $optional \in True, False$ is a flag describing whether the token is optional. For the purposes of learning templates, we will want to describe three types of tokens based on their alternative count:

- **Concrete Tokens** - $alt_count = 1$. A fixed field in a template; $alt_count = |value|$
- **Alternative Tokens** - $alt_count >= 1$. Field with a low variability; $alt_count = |value|$
- **Random Tokens** - $alt_count = -1$. Field in a template that is so variable, that keeping track of its values doesn't provide useful increase in information.

Category priority allows us to design token alphabets that have categories with patterns describing languages that are not mutually exclusive. For example, we can define a category $(.*, 0, 0)$ that matches any string, but at a low priority, as a sort of catch all category. This allows us to design robust programs that will be able to deal with unexpected inputs.

2.4 Edit Distance

This section has been adopted from [16]

Edit distance is a way of quantifying how dissimilar two strings are, by counting the minimum number of operations required to transform one string into the other. In other words, given the strings a and b on an alphabet Σ, the edit distance $d(a, b)$ is the minimum-weight series of edit operations that transform a into b.

Consider the following operations over strings:

- **Insertion** of a single symbol. Inserting symbol x into string $a = uv$ produces the string $b = uxv$.
- **Deletion** of single symbol. Deleting symbol x from a string $a = uxv$ produces the string $b = uv$.
- **Substitution** of a single symbol. Substituting symbol x for symbol $y \neq x$ in a string $a = uxv$ produces the string $b = uyv$.

[1] its edges point towards the root

- **Transposition** of two adjacent characters. Transposing two symbols $x \neq y$ in string $a = uxyv$ produces the string $b = uyxv$.

Note that the weight of these operations doesn't have to be uniform.

Different variants of edit distance can be obtained by restricting the set of edit operations:

- **Hamming** distance allows only substitution. It requires the compared strings to be the same length.

- **Levenshtein** distance allows deletion, insertion and substitution.

- **Damerau-Levenshtein** distance allows insertion, deletion, substitution, and transposition. Compared to Levenshtein distance, Damerau-Levenshtein distance allows for greater nuance when setting the weights of edit operations.

Chapter 3

Alignment

In this chapter, we are going to define the concept of sequence alignment and how we can use it to infer sequence generation patterns.

Sequence alignment is the process of identifying regions of similarity among sequences. It is most commonly used in bioinformatics for arranging DNA, RNA, or protein sequences, and finding similarities that may indicate functional, structural or evolutionary relationship between the sequences. However, sequence alignment can also be used to learn different information about the sequences, such as how similar their are, or what sequence of edit operations would need to be performed to make them identical. The tools developed for alignment in bioinformatics are not applicable for our purposes, as they always expect the existence of a finite alphabet of letters (such as nucleotides or amino acids). We are, however, dealing with an infinite alphabet of words. Many are also specifically optimized for aligning biological sequences, and implement non modifiable scoring systems.

There are two axes on which we can categorize alignment algorithms: number of sequences aligned, and locality of the alignment. *Pairwise* alignment methods align only two sequences, while *multiple sequence* alignment methods try to align all of the sequences in a given set. In this theses we will only discuss pairwise methods. *Global* alignment techniques attempt to align every letter in both all sequences. They are most useful when the sequences being aligned are similar and close in length. Global alignments can, however, end in gaps, and can, therefore, align sequences of different lengths. A different way of looking at global alignments is that they find the optimal (in respect to the scoring function) sequence of edits that have to be performed for the two sequences to match. This means that if the sequences are very different, the alignment will contain a high number of mutations; which is useful, as it allows us to find such sequences. *Local* alignment techniques attempt to find regions of high similarity in the sequences being aligned. They are most useful when aligning dissimilar sequences that are suspected to contain short regions of similarity that we are interested in.

In this paper, we will also refer to the result of an alignment algorithm as to an alignment. Any two sequences of equal length are an alignment. However, usually we are interested in an single *optimal alignment.* What is considered optimal depends on the alignment method and the scoring function used.

3.1 Spaces and Gaps

A *space* is the representation of a single insertion or deletion in an alignment.

A *gap* is an inserted or deleted substring in a string, i.e. it is a maximal consecutive run of spaces in a sequence. The *length* of a gap is the number of spaces (indel operations) in it [4].

Example 1. Consider the alignment:

```
a t t - - c a g g t t
a t c g t c a - - - t
```

This alignment has two gaps (denoted by the hyphen symbol), and a total of five spaces.

Gaps represent an indel operation. Whether the specific operation is a deletion or an insertion depends on how we interpret the sequences being aligned. Usually, we consider both sequences equivalent, and the difference in indel operations is lost.

3.2 Needleman-Wunsch

The Needleman-Wunsch algorithm [20] is dynamic programming algorithm for finding an optimal global alignment. It was designed for use in bioinformatics to align protein an nucleotide sequences, but can be used to align any two sequences.

In this section, the original version of the algorithm is presented. Both space and time complexity of this algorithm is $O(mn)$ where m and n are the lengths of the first and second sequence respectively. The time complexity can be improved to $O(mn/\log n)$ using the Four Russians method [19].

Let $A = a_1a_2 \dots a_n$ and $B = b_1b_2 \dots b_m$ be two sequences, where n and m are the lengths of the sequences to be aligned.

1. Determine the scoring system. We will discus different scoring strategies bellow, but for now, let

$$score(x, y) = \begin{cases} matchReward & \text{x and y match} \\ mismatchPenalty & \text{otherwise} \end{cases}$$

$$gapPenalty$$

   where $score(x, y)$ is the scoring match/mismatch function and $gapPenalty$ is indel score, which is used when a symbol aligns to a gap in the other sequence (representing a insertion or deletion).

2. Construct a scoring Matrix $M(n+1, m+1)$. Let $M_{(0,0)} = 0$

3. Fill the scoring matrix using:

$$M_{i,j} = max \begin{cases} M_{i-1,j-1} + score(a_i, b_j) \\ M_{i-1,j} + d \\ M_{i,j-1} + d \end{cases}$$

   The scoring matrix is filled iteratively, using values computed in previous steps. Values coming from the left and top represent gaps. The diagonal (top-left) neighbor represents an alignment of those two symbols, though it can be either a match, or a mutation.

4. Trace back the scoring matrix by starting at $M_{n+1,m+1}$ (the bottom right corner) and stepping through into the preceding (diagonal top left, top, or left) cell with the highest score, stopping at the top left corner.

Recording our path as we traceback will give us the optimal global alignment, with diagonal steps representing match/mutation, and lateral steps representing insertions or deletions.

A secondary traceback matrix can be computed along the scoring matrix, holding information about where each corresponding score cell got it's value from.

3.2.1 Example

This example will demonstrate the use of Needleman-Wunsch algorithm to align two sequences globally:

1. Let $s_1 = TGTGG$ and $B = TGACG$ be the sequences to be aligned

2. Let $match\_reward = 1, mismatch\_penalty = -1, gap\_penalty = -2$

3. Construct the score matrix M_s and the traceback matrix M_t

Table 3.1: Matrices after initialization

(a) Score Matrix

		T	G	T	G	G
	0					
T						
G						
A						
C						
G						

(b) Traceback Matrix

		T	G	T	G	G
	x					
T						
G						
A						
C						
G						

4. Fill the matrices

Table 3.2: Matrices after initialization

(a) Score Matrix

		T	G	T	G	G
	0	-1	-2	-3	-4	-5
T	-1	1	0	-1	-2	-3
G	-2	0	2	1	0	-1
A	-3	-1	1	1	0	-1
C	-4	-2	0	0	0	1
G	-5	-3	-1	-1	-1	1

(b) Traceback Matrix

		T	G	T	G	G
	0	←	←	←	←	←
T	↑	↖	←	←	←	←
G	↑	↑	↖	←	←	↖
A	↑	↖	↑	↖	↖	↖
C	↑	↑	↖	↖	↖	↖
G	↑	↑	↖	↖	↖	↖

5. Traceback to get the alignment: Starting at the bottom right cell we follow arrows in M_t (representing source of the score in that particular cell) until we hit the top left cell

```
T G T G G
T G A C G
```

3.3 Smith-Waterman

Smith-Waterman algorithm performs *local* sequence alignment[24]; that is it compares segments of all possible lengths. It is a variation of the Needleman-Wunsch algorithm (described above), the main difference being that Smith-Waterman sets negative scoring matrix cells to zero, and that the scoring begins at the cell with the highest score, instead of the bottom-right cell. This renders the local alignments visible. This modification allows the Smith-Waterman algorithm to find regions with high similarities in the two sequences being aligned.

3.3.1 Example

This example will demonstrate the use of Smith-Waterman algorithm to align two sequences locally:

1. Let $s_1 = TGTGG$ and $B = TGACG$ be the sequences to be aligned
2. Let $match\_reward = 2, mismatch\_penalty = -2, gap\_penalty = 1$
3. Construct the score matrix M_s and the traceback matrix M_t
4. Fill the matrices M_s and M_t

Table 3.3: Filled Matrices

(a) Score Matrix

		T	G	A	G	C
	0	0	0	0	0	0
T	0	2	1	0	0	0
G	0	1	4	3	2	1
T	0	2	3	2	1	0
G	0	1	4	3	4	3
G	0	0	3	2	5	4

(b) Traceback Matrix

		T	G	A	G	C
	0	0	0	0	0	0
T	0	↖	←	←	0	0
G	0	↑	↖	←	↖	←
T	0	↖	↑	↖	↖	↖
G	0	↑	↖	←	↖	←
G	0	↑	↖	↖	↖	←

5. Traceback to get the alignment: Starting at cell with the highest score ($M_{s(5,4)} = 5$) we follow arrows in M_t (representing source of the score in that particular cell) until we hit 0.

```
T G - G
T G G G
```

3.4 Parameters of Alignment Algorithms

In this section, we will explore different parameters of the two alignment algorithms described above.

3.4.1 Gap Penalty

Gap penalty is a cost score that is assigned to a gap (representing an insertion or deletion) in an alignment [13]. Gap penalty allows us to reveal insertions or deletions in an alignment.

Two commonly used values when talking about gap penalty are *gap opening penalty* and *gap extension penalty*, the former being the score for opening a new gap, the latter being the score for extending an existing gap by one space. Note that these values are usually a positive number, and alignment algorithms substract them from the final score, resulting in a penalty (see Section 3.2 and Section 3.3).

Different gap penalty scoring strategies allow us to incentivize different kinds of gaps.

Constant Gap Penalty

The simplest choice for a gap penalty is the constant gap penalty model, where each gap is given a constant weight of W_g, independent of it's length. Each individual indel operation is therefore free, we only score the existence of such operation [4].

Constant gap penalty can be implemented by setting *gap opening penalty* to a positive number, W_g and *gap extension penalty* to zero, and is a special case of the *affine gap penalty*.

This model encourages the algorithm to make fewer, larger gaps.

Linear Gap Penalty

The *Linear gap penalty model* considers only the length of the gap to determine it's overall weight. Each individual indel operation contributes the same amount to the total penalty. For a gap of length q it would be:

$$W_g = qW_s$$

Linear gap penalty can be implemented by setting both *gap opening penalty* and *gap extension penalty* to W_s, and is a special case of the *affine gap penalty*.

This model encourages the algorithm to make smaller, more frequent gaps.

Affine Gap Penalty

The *Affine gap penalty model* scores both a gap as a whole and it's length. The weight (the total penalty) of a gap of length q is:

$$W_g = W_{open} + qW_{extend}$$

(The model is called „affine“ after this affine formula.)

Affine gap penalty can be implemented by setting both *gap opening penalty* to W_{open} and *gap extension penalty* to W_{extend}. Depending on the ratio $\frac{W_{open}}{W_{extend}}$ this model will either encourage fewer, larger gaps ($W_{open} \gg W_{extend}$, very common in biology (see Section **??**))) or more frequent, small gaps ($W_{open} \ll W_{extend}$).

Absolute Value of the Gap Penalty

Often, it is not clear what the absolute values of *gap opening penalty* and *gap extension penalty* should be, as it depends on both the purpose of the alignment and other parameters, such as score or length of the sequence. It is obvious that higher values will result in more closely related matches, with fewer gaps. On the other hand, a lower gap penalty will allow us to find matches that are more distant.

3.4.2 Substitution Scoring Function

Substitution scoring function $s(a,b)$ assigns two elements of the sequences being compared a score. This score represents how likely it is those elements would be substituted for one another. In biology, this usually means how likely one element is to mutate to another.

The simplest approach is to return a positive score when the elements match exactly, and a (equal) negative score when they don't.

$$s(a,b) = \begin{cases} 1 & a = b \\ -1 & otherwise \end{cases}$$

Another common approach is a *scoring matrix*, which is a matrix in which each element represents the score of matching (replacing) one element with another [14]. Scoring matrices are commonly used in bioinformatics for aligning protein and nucleotid sequences. Examples of such matrices are BLOSUM62 [7] for amino acids and DNAfull [14] for nucleotids. Both were computed by analyzing the frequencies of substitutions in collections of known alignments.

Scoring matrices are used by many implementations of alignment algorithms for biological sequences such as FASTA [12] and BLAST [6].

Chapter 4

SMS and SMS spam

4.1 SMS

SMS, or *short message service*, is a text messaging service component of most telephone systems. It uses standardized communication protocols that let mobile devices exchange short text messages. The SMS was included in the GSM (Global System for Mobile Communications) standards from the beginning. There are two types of message described: Mobile Originated (MO), meaning the message was send *from* a mobile handset to another mobile handset, and Mobile Terminated (MT), meaning the message was sent *to* a mobile handset and originated from another handset or from a software application.

SMS is realised by the use of the *Mobile Application Part* (MAP) of the *SS7* protocol, with SMS protocol elements being transported across the network as fields within the MAP messages [3]. Messages are sent with the MAP MO-ForwardSM and MT-ForwardSM operations, whose payload length is limited by the constraints of the signaling protocol to precisely 140 bytes. SMS can be encoded using three alphabets: the default *GSM 7-bit* alphabet, the *8-bit data alphabet*, or the 16-bit *UCS-2* alphabet [1]. Choice of the alphabet leads to the maximum individual message size of 160 7-bit characters, 140 8-bit characters, or 70 16-bit characters. GSM 7-bit support is mandatory for GSM handsets and network elements [1]. Longer content can be sent using multiple messages (for example with a *User Data Header* [2] in the front) and then concatenated on the receiving end into a single message. However, for the purposes of this thesis, we will only consider single messages up to 160 characters in length.

4.2 Spamming

Spamming is the act of sending numerous unsolicited messages (spam) to a large number of recipients, often for the purposes of advertising. Usually (and for the purposes of this paper always) spamming refers only to the use of electronic communication systems to send such messages.

The precise definition of spam is dubious. OECD (Organization for Economic Co-operation and Development) aggregated several definitions and discussions of spam, and summarized characteristic properties of spam. They split them into two categories - *primary* and *secondary*. The *primary* characteristics include unsolicited electronic commercial messages, sent in bulk. The *secondary* characteristics are ones that are frequently associated with spam, but perhaps not necessary [21]. Table 4.1 shows those characteristics.

Table 4.1: Primary and secondary characteristics of spam

Primary characteristics	Secondary characteristics
Electronic message	Uses addresses collected without prior consent or knowledge
Sent in bulk	Unwanted
Unsolicited	Repetitive
Commercial	Untargeted and indiscriminate
	Unstoppable
	Anonymous and/or disguised
	Illegal or offensive content
	Deceptive or fraudulent content

The Spamhaus Project defines spam simply as *unsolicited bulk email*, considering spam to be „an issue about consent, not content“ [22]. This distinction is important, because when legislators attempt to regulate the content of spam messages, they come up against free speech issues. This makes creating anti-spam legislation difficult to implement effectively. In fact, most spam originates in countries with lax anti-spam laws [23].

4.3 SMS Spam

With the advances in effective filtering and user awarness, email spam return is diminishing. Traditional email spammers are moving to mobile networks. The SMS service is attractive for criminals for a number of reasons. In contrast with phone or email communication, users are able to interact with SMS services in nearly every environment, without disrupting those around them. SMS also leads to near instantaneous interaction with the recipients: SMS marketers claim SMS messages open rates are higher than 90% and are opened within 15 minutes of receipt. Contrast that to the open rate in email of only 20-25% withing 24 hours of receipt [8]. In addition, SMS is often considered a trusted service, and users are more comfortable using it for confidential information exchange. This can result in a higher response rate to SMS spam than to email spam [10]. It is also becoming cost effective to target SMS because of the availability of unlimited pre-pay SMS packages.

Another issue of SMS is that actual spam is not the only troublesome form of bulk communication, from the provider's point of view. Private companies often use subscription plans designed for regular customers to avoid using more expensive means designated for commercial purposes. Such communication can be solicited and important for the recipient, but it's bulk nature represents an increased, uncompensated strain on the network, as well as a loss in revenue.

4.4 Spam Campaigns

The term "spam campaign" is commonly used with varying degrees of generality to mean anything from all spam of a certain type (e.g. pharmaceutical), through spam intended for a specific purpose (e.g. phishing) to spam continuously generated from a single template [18]. For the purposes of this work, we will refer to spam campaign as to a set of messages which were generated from the same template. A template is a description used to craft individual spam messages. It is composed of both fixed text and variable fields, which are expanded to generate a single message. Their use allows the spam message to be personalized (it could, for example, include the recipients name).

```
Your A/C No:{NUMBER} has been {"credited"|"debited"} Tk.{AMOUNT} on
        {DATE} by {"transfer"|"cash"}. Current Balance Tk.{AMOUNT}. Thank You,
        R****i Bank Ltd.
Dear Client, TK. {AMOUNT} credited to {NUMBER} by {"Transfer"|"On−line
        cash"} dated {DATE} {TIME}. Remaining balance {AMOUNT}. MTB Helpline
        ****9
```

There are three interesting things to note:

1. The template is composed of three types of components:
   - **Fixed words**, which will always appear in the same spot in each message. For example „Your“ or „Balance“,
   - **Variable**, which are a choice from a finite set. For example „credited“|„debited“,
   - **Highly variable**, which are generated from a very large (sometimes random) set. For example NUMBER being one of many account numbers.
2. Some strings that might be variable or highly variable in one template might be constant in another. For example, the word „credited“ appears in both templates, but it can be replaced by „debited“ in one of them. Likewise, all numbers in the first template are variable, but there is a constant number in the second template.
3. A single template might produce messages with different number of words. This is important, because a simple leftmost alignment (aligning messages by words from the first word) will perform poorly, as a single word shift will throw it off. This problem is easily dealt with by introducing the concept of gaps (see Section 3.1).

The variable nature of spam messages makes them harder to be grouped with messages belonging to the same campaign (generated from the same template). Grouping messages together is important: not only does it allow us to approximate the size of the attack, but if we can learn a template from which those messages were generated, we can use it to filter further spam messages from the same campaign.

4.5 SMS Spam detection in Mavenir s.r.o.

Mavenir s.r.o. uses text clustering to detect SMS spam. Messages are assigned to clusters (representing campaigns) based on text similarity.

Mavenir currently uses a combination of *locality-sensitive hashing* (LSH) and pre-screening that requires a partial match, to calculate text distance and cluster messages.

LSH [17] is an algorithmic technique that assigns similar input items into the same *buckets* (assignes them the same hash) with high probability. The number of such buckets will be much smaller than the universe of possible input items. LSH differs from conventional hashing techniques (such as ones used in data storage and retrieval) in that hash collisions are maximized, not minimized. Therefore, LSH can be considered a data clustering method.

An *LSH family* $\mathcal{F}$ is defined for a metric space $\mathcal{M} = (M, d)$, a threshold $R > 0$ and an approximation factor $c > 1$. This family $\mathcal{F}$ is a family of functions $h : M \to S$ which map elements from a metric space to buckets $s \in S$. The LSH family satisfies the following conditions for any two points $p, q \in M$, using a function $h \in \mathcal{F}$ which is chosen uniformly at random:

- if $d(p, q) \leq R$, then $h(p) = h(q)$ with probability at least P_1
- if $d(p, q) \geq cR$, then $h(p) = h(q)$ with probability at most P_2

A family is interesting when $P_1 > P_2$. In other words, only functions that assign two samples with distance smaller than some threshold the same hash more often than to samples with distance greater than the threshold are of interest to us when designing LSH.

Pre-screening is a method used to choose candidates for comparison. A set of mandatory tokens is chosen for each existing campaign. The message being compared has to contain a certain number of these tokens before it's even considered for the particular campaign. A bigger match typically hints at a better candidate. The process of choosing the mandatory tokens is non-trivial, and beyond the scope of this work.

4.5.1 Fragmentation and Merging

There are two problems we need to contend with when using the approach described above to sort messages into campaigns: excessive *merging* and *fragmentation*.

Excessive merging happens when messages that were in reality generated from different templates are merged into one campaign. This type of error is not a too common in practice, mostly because of the pre-screening mechanism described above.

Fragmentation happens when messages that were in reality generated from one template are classified into different campaigns. This type of error is, unlike excessive merging, very common, and happens often in campaigns that we are interested in (spam campaigns), as attackers are intentionally increasing variability of their campaigns by introducing noise into their spam messages. In other words, the campaigns that are the most important to detect are the most vulnerable to fragmentation when using the method described above.

One solution to the problem of fragmentation is to use multiple clustering methods. Diversification is a common and effective method in combating fraud, as it is harder for an attacker to overcome multiple simple detection methods than one perfect method. Chapters 5 and 6 of this paper focuses on describing one such method.

4.5.2 Data provided by Mavenir s.r.o.

For the purposes of developing an algorithm for inferring spam SMS generation templates Mavenir s.r.o. provided a sample dataset of SMS spam. The data consists of messages and campaign keys assigned to them by the method described above (4.5). The dataset contains a total of 58,450 messages.

Example 2. Four rows from the dataset:

```
1   4E03003630044A810020110200A0216040208072023086002000A01400821E0001188420,
        "Tu codigo es 49158. Ve a Facebook e introducelo para confirmar. #fb"
2   4E03003630044A810020110200A0216040208072023086002000A01400821E0001188420,
        "Tu codigo es 67814. Ve a Facebook e introducelo para confirmar. #fb"
3   4E030058208B032958B00301000E2E70121230CC800188EC4094A0C0010B4DCC00081083,
        "Dear Client, TK. 20776.00 credited to ***82520 by Transfer dated
        2020-09-08 16:16. Remaining balance 177804.08. MTB Helpline ****9"
4   4E030058208B032958B00301000E2E70121230CC800188EC4094A0C0010B4DCC00081083,
        "Dear Client, TK. 100000.00 credited to ***98648 by On-Line Cash
        dated 2020-09-08 3:58. Remaining balance 3016218.73. MTB Helpline
        ****9"
```

Rows one and two were classified to the same campaign, rows three and four likewise.

Sensitive data, such as names, addresses, and account numbers have been replaced by placeholders.

Example 3. A message with account number censored (campaign key omitted):

```
"Your A/C No:3251*****7543 has been DEBITED Tk.22,500.00 on 08−SEP−2020 by CASH. Current Balance Tk.596. Thank You, R****i Bank Ltd."
```

The data was chosen so that most prior campaigns were not significantly fragmented or excessively merged, however, some fragmentation is present. This is good, as it provides an opportunity to evaluate whether our algorithm manages to avoid the same mistakes the LSH algorithm makes. It, however, also means that we can not consider the data correctly labeled for the purposes of using automatic learning algorithms.

Example 4. Fragmented campaign - these two messages are obviously generated from the same campaign, however, a different has been assigned to them.

```
4E0300582...0081083,"Dear Client, TK. 1.41 credited to ***47503 by Transfer dated 2020−09−08 16:16. Remaining balance 121.40. MTB Helpline ****9"
4E03004D3...A082083,"Dear Client, TK. 909.06 credited to ***20918 by Transfer dated 2020−09−08 16:16. Remaining balance 2628.92. MTB Helpline ****9"
```

Chapter 5

Algortihm for Inferring Sequence Generation Patterns

In this chapter, we are going to describe the proposed algorithm for iteratively inferring sequence generation templates. In this chapter, we will describe the algorithm generally, and define a number of *hyperparameters* that will control how the algorithm learns templates. Their precise values will differ depending on the nature of the data being classified. 6 describes how we arrived at values for classifying data in Section 4.5.2.

First, we will need an algorithm that takes a string to classify, a database of campaigns, and a number of parameters, classifies the string to a campaign (existing one, or a new one), updates the campaigns to reflect the information learned from this new string, then returns the ID of the campaign to which the new string was classified and a database of the updated campaigns. The general idea as follows:

1. Tokenize the new string.
2. Pick a campaign from a database. If there are no campaigns left, create a new one with a pattern equal to the new string.
3. Attempt to match the new string against the standing pattern of a campaign. If it matches, assign the string to the campaign, and terminate.
4. Align the new string with the standing pattern. If the alignment doesn't conform to parameters (it is too short, the align score is below the threshold...), return to step 2
5. Merge the alignment with the standing pattern, to update the pattern. If the new pattern doesn't conform to parameters (there are too many random tokens...), return to step 2, otherwise terminate.

We will then need to expand this algorithm to take a stream of strings and iteratively build a database of campaigns from scratch.

5.1 Classification Algorithms

In this section, we will look at how the proposed algorithms could be implemented. Two algorithms will be described: *streamClassify*, which will take a stream of strings, and sort them into campaigns, and *classify*, which will take a single string, and return a campaign to which it belongs. Note that both algorithms share a number of *hyperparameters*:

- Tokenize - a function used to split a string into a sequence of tokens
- Align - a function that takes two sequences of tokens, and returns their optimal alignment
- Score - a function to score the similarity of two tokens. Identical to the scoring function described in Section 3.4.2
- CollapseAlignment - a function that takes two sequences of tokens (ostensibly the result of aligning two sequences of tokens), and returns one sequence by collapsing them.
- JudgeAlignment - a function that evaluates an alignment (two sequences of tokens and their score), and decides whether it is satisfactory. This will be often achieved by measuring several properties of the alignment, such as its length or its score, and comparing them to some boundaries.

These are used to control the process of learning campaign templates. How well (or if at all) the algorithm will perform depends greatly on the choice of these hyperparameters. They will be described in greater detail below.

The *streamClassify* algorithm takes a stream of strings, and classifies them into campaigns as they come. This algorithm can be used to build a campaign database from scratch.

```
Input:
        stream (stream of strings),
        Tokenize (A~function to split a string into a sequence of tokens
            ),
        Align (A~function to compute alignment),
        Score (Substitution scoring function),
        CollapseAlignment (A~function that colapses an alignment into a
            regular expression),
        JudgeAlignment (A~function that returns true if an alignment
            that is passed to it is satisfactory, or to false otherwise)
        gapOpenPenalty (Smith-Waterman gap opening penalty),
            gapExtendPenalty (Smith-Waterman gap extension penalty)

Output:
        Database of campaigns
=========================
CampaignDb campaigns
while there stream is not empty:

        Sequence s~:= tokenize(next string)

        id, campaigns := classify(s, campaigns, Align, Score,
            CollapseAlignment, JudgeAlignment)

return campaigns
```

Algorithm 5.1: streamClassify

Algorithm *classify* takes a sequence of tokens, a database of campaigns, and a number of parameters, and classifies the sequence to a campaign, i.e. it finds a campaign with a template that either matches the sequence, or aligns with it satisfactorily (based on the parameters), then returns the id of this campaign and updates the database of campaigns. If no satisfactory campaign has been found, the algorithm creates a new one.

```
Input:
        s~(sequence of tokens),
        db (database of campaigns),
        Align (A~function to compute alignment),
        Score (Substitution scoring function),
        CollapseAlignment (A~function that collapses an alignment into a
            regular expression),
        JudgeAlignment (A~function that returns true if an alignment that is
            passed to it is satisfactory, or false otherwise)

Output:
        id (id of the campaign to which we classified the input sequence s)
        db (updated database of campaigns)
        =========================
foreach id, Campaign c in db:
        if c.match(s):
                return id, db
        else:
                alignS, alignC, score := Align(s, c, Score)

                align := CollapseAlignment(alignS, alignC)

                if JudgeAlignment(align, s, c, score):
                        db[id] := align
                        return id, db
                else:
                        continue

-- create new campaign
db[id := nextId(db)] :=
s~return id, db
```

Algorithm 5.2: classify

5.2 Tokenization

The first step in the classification algorithm is to tokenize the string being classified, meaning to transform it into a sequence of tokens.

We define function $tokenize(split, \Sigma_T, s)$, where s is the string to tokenize, $split$ is a unary function that takes a string and returns a sequence of strings, and Σ_T is a token alphabet(2.3): [1]

$$tokenize(split, \Sigma_T, s) = [(head(category(s', \Sigma_T)), \{s'\}, 1, False) | s' \in split(s)]$$

Function *tokenize* takes a string, splits it into a sequence of strings using function *split* and constructs a sequence of tokens. Each token will be initially very similar. The value of

[1] $category(s', \Sigma_T)$ is a set, not a collection, therefore operation *head* doesn't, in theory, make much sense, however, in practice, most implementations will be working with a collection. Those that won't, will have to define an operation that chooses a member of the set uniformly at random in place of *head*

a new token will be composed of only s'. Its category will be one with the highest priority that matches s'. Its alternative count will always be equal to zero, and it will never be optional.

Note that if there exists a s' that isn't matched by any category in Σ_T, the result of *tokenize* is not defined, and should be treated as an error. Barring that, the behavior of *tokenize* will depend mostly on the value of *split*. The token alphabet Σ_T doesn't affect the shape of the resulting sequence, nor the values of tokens in the sequence. The trivial alphabet $\Sigma_{T_0} = \{(.*, 0)\}$ can always be used, if categorization of tokens is not desirable. As we will see later, however, categories give us a great deal of power over how sequences will be aligned.

There are two ways of thinking about the *split* function: it defines what is a token looks like; it defines what text doesn't provide useful information for classification. In other words, *split* can be defined either in terms of what text forms tokens, or in terms of what text separates tokens. However, not all *split* functions will be describable in both ways.

In most cases, the *split* function and token alphabet Σ_T will be the same for every invocation of *tokenize* during one run of classifications, therefore, it might be a good idea to partially apply the function *tokenize* and produce a function $tokenize'(s)$.

5.2.1 Token Types

As described in Section 2.3, tokens are divided into three types based on their alternative count: concrete tokens, alternative tokens, and random tokens. In this section we will take a closer look on what these types mean and how they can be used in learning templates.

Concrete tokens represent fixed spots in a template. Their values are always literals. All tokens in the sequence *tokenize* returns will be concrete tokens.

Alternative tokens represent variable parts of a template: spots that can have several different values. Those can be, for example, places where the attacker used synonyms, such as „`install`" and `download`„, to introduce variability into their spam campaign to throw off spam detectors, without losing semantic meaning for the human recipient. Alternative tokens are created when two concrete tokens are aligned. When being joined, the alternative count of the resulting token is automatically incremented (see Section 5.5 for more information on how alternative tokens are joined).

Random tokens represent spots where the template is so variable that keeping track of the concrete values provides only a marginal increase in information gained (quality of alignment) over accepting any value. Those can be names, telephone numbers, verification codes, addresses... We can see that while concrete telephone number is not of much interest to us when learning a template, the fact that a telephone number appears in a specific spot is very important. We can therefore augment random tokens with *token categories*, to allow us to retain such information.

Example 5. Consider the following messages from the same prior campaign:

```
G−2***63 is your Google verification code.
7***21 is your Skrill authentication code.
4***12 is your NETELLER authentication code.
```

an example of template that could generate such messages could be:

```
(("\w*)?\d+", 0), .*, −1), (("w+", 1), is, 1), (("w+", 1), your, 1),
    (("w+",1),Google|Skrill|NETELLER, 3), (("w+", 1),
    verification|authentication, 2) (("w+", 1), code, 1)
```

Notice that the words `Google, Skrill, and NETELLER` appear in all of them in the same spot, and so do the words `verification and authentication`, so those are a good candidates for alternative tokens. At the beginning of the message is a number. We can conclude that this number will be different for every message (in this case by interpreting the semantics of the message, but during automatic classification by the fact that the alternative count of the token in that spot increases too much), so we declare it random. We can, however still assign it a category, so that it won't match messages that begin with the word `hello` for example. The other spots always contain the same words, so we declare them concrete.

5.3 Campaigns

A *campaign* c is a pair $(id, template)$ where id is a identification of the campaign (for example, a natural number), and *template* is a sequence of tokens.

To decide whether a message belong to a campaign, we need to be able to determine whether the campaigns template could generate it. To do so, we will construct a regular expression out of the template in the following manner:

1. Construct a regular expression out of each token t in the template
   (a) If t is concrete or variable
      - if t is optional → $((\bigcup t.value)?)$ [2]
      - else → $(\bigcup t.value)$
   (b) If t is random → $(.*?)$
2. Join groups constructed in the previous step by a sequence (by a single space, for example)

During tokenization, characters that were present in the original string are lost. It is not possible to compare match a raw message with the regular expression constructed from a campaign's template. It first has to be tokenized, and the values of the tokens (which will all be concrete) have to be joined by the same sequence that was used to join the campaign. Now, the message is ready to be matched with the campaign. A string is said to belong to a campaign if it belongs to the language that the regular expression describes.

5.4 Scoring

To perform an alignment, we need a way to calculate how likely one token is to be substituted for another in an alignment. For this, we need a *substitution scoring function* (see Section 3.4.2). The substitution score in biological sequence alignment represents the likelihood of a mutation. The substitution score in our algorithm represents how likely it is that two tokens were generated by a single field in a template. This needs to be taken into consideration when constructing a scoring function

We define a binary function *score* that takes two tokens, and returns a real number. More formally: Let Σ_T be a token alphabet and L_c a language denoted by the pattern of

[2] $\bigcup S$, where S is a set of regular expressions, denotes a regular expression that describes a language that is a union of all languages described by all regular expressions in S

category $c \in \Sigma_T$. Then $tokens(c) = \{(c, s) | s \in L_c\}$[3] is a set of all tokens that could match category c, and $\mathbb{T}_{\Sigma_T} = \{tokens(c) | c \in \Sigma_T\}$ is a set of all tokens matching a category in Σ_T. Then we define function *score* as

$$score : \mathbb{T}^2_{\Sigma_T} \to \mathbb{R}$$

The precise behavior of this function can vary greatly based on the types of sequences being aligned. However, one universally useful attribute is the ability of *score* to return high values when tokens are likely to have been aligned, and low values when they are unlikely to be aligned.

Now we will explore some possibilities for function *score*. These techniques are not mutually exclusive, and multiple off them can be used when construction the *score* function For conciseness, we define the following constants:

- $c_0 = (.*, 0)$,
- $word = (\backslash w+, 1)$,
- $num = (\backslash d+, 1)$,

5.4.1 Match/Mismatch

The simplest way to implement function *score* is to assign it a positive value n if the values of tokens match, and a negative value $-m$ if they don't

$$score(t1, t2) = \begin{cases} n & t1.value = t2.value \\ -m & \text{otherwise} \end{cases}$$

The biggest disadvantage of this approach is that it is susceptible to being “easily fooled„ by noise. Consider the following alignment:

$(c_0,$ `Quick)` $(c_0,$ `brown)` $(c_0,$ `fox)` $(c_0,$ `jumps)` $(c_0,$ `over)` $(c_0,$ `dog)`
$(c_0,$ `Quickk)` $(c_0,$ `brownn)` $(c_0,$ `foxx)` $(c_0,$ `jumpss)` $(c_0,$ `overr)` $(c_0,$ `dogg)`

Corresponding tokens in this alignment differ only in one letter. Under the Match/Mismatch scoring strategy, this alignment would have a score of $-6m$, even though we can see the sentences are identical semantically, barring a small amount of noise.

5.4.2 Edit distance

A more complicated approach is to score substitutions based on their text similarity. This way, small changes will have a small impact, making the alignment more resistant to noise. This will naturally come at a cost to speed.

Example 6. Let n be a match reward, and $edit\_distance(s1, s2)$ a function that calculates the edit distance of two strings (see Section 2.4). Then

$$score(t1, t2) = n - \frac{n}{\frac{|t1.value| + |t2.value|}{4}} \cdot edit\_distance(t1.value, t2.value)$$

By including length of the strings in the calculation, we ensure the score is moves linearly from n to $-n$, regardless of how long the string are.In this example, when the sequences

[3] Alternative count and optionality of a token are omitted for conciseness, as they will not be important

are identical (have an edit distance of 0), they have the highest score (n), when their edit distance is equal to the mean average of their lengths, the score is equal to $-n$, and when it is even higher, the score is even lower. This allows us to better predict the score, and also ensures that longer words will not outperform shorter ones.

$$score(t1, t2) = \begin{cases} n & edit\_distance(t1.value, t2.value) = 0 \\ -n & edit\_distance(t1.value, t2.value) = \frac{|t1.value|+|t2.value|}{2} \\ < -n & edit\_distance(t1.value, t2.value) > \frac{|t1.value|+|t2.value|}{2} \end{cases}$$

5.4.3 Substitution Matrix

Substitution matrix M is a matrix of shape $(\mathbb{T}_{\Sigma_T} \times \mathbb{T}_{\Sigma_T})$. Each element of matrix M represents the substitution score of two tokens. Let $\mathbb{T}'_{\Sigma_T}$ be an indexable sequence created by arranging elements of $\mathbb{T}_{\Sigma_T}$ in arbitrary order. Then $M[i, j]$ is the substitution score of elements $\mathbb{T}_{\Sigma_T}[i]$ and $\mathbb{T}_{\Sigma_T}[j]$

$$score(t1, t2) = M[index(t1, \mathbb{T}_{\Sigma_T}), index(t2, \mathbb{T}_{\Sigma_T})]$$

Where $index(t, s)$ is the index of element t in sequence s.

Example 7. Let $\Sigma_T = (a, 0), (b, 0), (c, 0)$. Then obviously $\mathbb{T}'_{\Sigma_T} = [((a, 0), a), ((b, 0), b), ((c, 0), c)]$. We can define substitution matrix M as

$$M = \begin{bmatrix} 1 & -1 & -1 \\ -1 & 1 & -1 \\ -1 & -1 & 1 \end{bmatrix}$$

Note that a *score* function defined using this matrix will be identical with the function defined in Section 5.4.1 for $n = m = 1$.

Substitution matrix is a very fast implementation of scoring function, as it has a $O(1)$ complexity, and it's speed depends only on the speed of accessing the array elements. However, it requires a known and finite alphabet of tokens. This method is therefore not well suited for our purposes, it is, however, a very popular method in bioinformatics.

5.4.4 Categories

In dna or protein sequence alignment, substitution score of different bases/acids represents the likelihood of those elements to be substituted for one another. We can implement a similar approach using token categories, since their number will be finite and they will be known in advance.

Example 8. Function *score* that penalizes alignment of tokens with different categories

$$score(t1, t2) = \begin{cases} n & t1.category = t2.category \land t1.value = t2.value \\ -m & t1.category = t2.category \land t1.value \neq t2.value \\ -2m & \text{otherwise} \end{cases}$$

Example 9. Function *score* that does not take into account the values of tokens of certain categories Let $\Sigma_T = c_0, word, num$. Then:

$$score(t1, t2) = \begin{cases} n & t1.category = t2.category \land t1.category = num \\ n & t1.value = t2.value \\ -m & \text{otherwise} \end{cases}$$

This approach is useful when dealing with token categories that we expect to be highly variable, such as phone numbers.

5.5 Alignment Collapsing

An alignment is a pair of sequences of equal length. To construct a template from them, we first need to collapse them. Let $A = (A_s, A_c, score)$ be and alignment. A_s is the part of the alignment extracted from the message being classified, and A_c is the part of the alignment extracted from the campaign we are attempting to classify the message into. Algorithm 5.3 collapses alignment into a template

```
Input: $A_s$, $A_c$, maximumAlternatives
Output: A~sequence of tokens

result := an empty sequence
for $t_s$, $t_c$ in zip($A_s$, $A_c$):
        if $t_s$ is a gap $\vee$ $t_c$ is a gap:
                $t$ := $t_s \cup t_c$
                make $t$ optional
        else
                if $t_s$ and $t_c$ don't have the same category:
                        $t$ := $t_s \cup t_c$
                        set $t.category$ to the closest common upstream node
                            of both categories
                else:
                        $t$ := $t_s \cup t_c$

        if alternatives in $t$ $>$ maximumAlternatives:
                make $t$ random

        append $t$ to result

return result
```

Algorithm 5.3: Collapse Alignment

5.6 Alignment Judging

Not all alignments are suitable to be templates. The alignment algorithm finds the optimal alignment of two sequences, but sometimes, if the sequences are too different, even that is not usable as a template. Alignment judging is the process of deciding whether the optimal alignment can be considered a template. This step makes our algorithm truly a classification algorithm. Without it, all messages would be classified into the same campaign.

The first step to construction an alignment judging function is to decide what an acceptable campaign template looks like. This was discussed in Section 4.4. By exploring available datasets, we determined, than the number of fixed (constant) fields is significantly higher that the number of variable and highly variable (random) fields. We also determined, that if a gaps were to occur in an alignment, i.e. a variable field in a template generated more than one token, they would be short, only one or two tokens long.

The precise form the alignment judging function will take also depends on the alignment algorithm and substitution scoring, and we will propose various alternatives in 6.

5.7 Metrics

In this section we will describe metrics for measuring the performance of our algorithm. The goal of the algorithm is to classify messages into campaigns, without introducing a significant amount of fragmentation (classifying messages from the same campaign into multiple) or excessive merging (classifying messages from different campaigns into one campaign). To evaluate how well our algorithm performs these tasks, we can compare its results to a reference classification. In this section, we will refer to campaigns from which messages were generated as *prior campaigns* and to campaigns to which we classified messages as *learned campaigns*.

5.7.1 Fragmentation rate

Fragmentation is the assignment of two strings that were generated from the same prior campaign into two different learned campaigns. Let c be a learned campaign, S a set of strings, P a set of prior campaigns, $p(s)$ a prior campaign from which the string $s \in S$ was generated. Learned campaign c is *fragmented* with respect to a set of strings S if $fragmented(c, S)$ is true:

$$fragmented(c, S) = \exists s \in S, p(s) \neq classify'(s)$$

We define a function $fragments$ that will show us all learned campaigns to which messages from each prior campaign were classified: a set of fragments of each prior campaign.

$$fragments(classify, S, P) = \{(c_p, \{Classify(s) \mid s \in S \wedge p(s) = c_p\}) | c_p \in P\}$$

or, in other words, $fragements(classify, S, P)$ is a set of sets of campaigns to which strings $s \in S$ which were generated from prior campaigns c_p were classified for each prior campaign $c_p \in P$.

We can now calculate how much algorithm $classify$ fragments. [4]

$$fragmentation\_mean(classify, S, P) = mean(\{|A| \mid (\_, A) \in fragments(classify, S, P)\})$$

$$fragmentation\_deviation = std(\{|A| \mid (\_, A) \in fragments(classify, S, P)\})$$

These values give us a high level idea of how the algorithm performs. $fragmentation\_mean$ is the average number of fragments a campaign is broken into. Ideally, it would be equal to 1, however, in cases where this is not the case, $fragmentation\_mean$ alone doesn't provide any information about how to improve the algorithm. $fragmentation\_deviation$ likewise is not very helpful: it can only hint on whether a high $fragmentation\_mean$ is caused by an outlier.

To diagnose the algorithm, we will have to look at the data at a less aggregate level. Additionally, we will require additional piece of information: "how much„ of the prior campaign is part of the campaigns to which it was fragmented.

$$A(t) = [classify(s) \mid s \in S \wedge c_p = p(s)]$$

[4] A colon is used for set comprehension notation in this case, to distinguish it from the set cardinality operator

$$fragments'(classify, S, P) = \\ [(c_p, c, |\{c' \mid c' \in A(c_p) \wedge c' = c\}|) \mid (c_p, c) \in P \times \{classify(s) \mid s \ \in S\}]$$

Function $fragments'(classify, S, P)$ constructs a sequence of elements $(c_p, c, count)$, where c_p is a prior campaign, c is a learned campaign and $count \in \mathbb{N}_0$ is the number of strings $s, P(s) = c_p$ (strings that were generated from a prior campaign c_p) that were classified into learned campaign c.

Let $sort(A, key)$ be a function that returns a sequence which has the same elements that a set A has, but sorted descending by values of function $key : a \in A \to \mathbb{Z}$. Then:

$$fragments\_by\_size(classify, S, P) = sort(fragments(classify, S, P), \lambda(\_, a).|a|)$$

$$fragments\_by\_deviation(classify, S, P) = sort(\\ [(t, std([count \mid (c'_p, \_, count) \in fragments'(classify, S, P) \wedge c'_p = c_p]) \mid c_p \in P], \lambda(\_, a).a)$$

$fragments\_by\_size$ shows us prior campaigns that are fragmented into many different campaigns.

Example 10. Prior campaign c_{p1} is fragmented into many campaigns:

$$(c_{p1}, \{c1, c2, c3, c4, c5\})$$

Prior campaign c_{p2} is fragmented into few campaigns:

$$(c_{p2}, \{c1, c2\})$$

Note that $fragments\_by\_size$ loses information about what “size„ (how many strings were classified into which campaign) the fragments are; it only cares about how many campaigns the prior campaign is fragmented into.

$fragments\_by\_deviation$ shows us prior campaigns that may or may not be fragmented into many campaigns, but whose fragments are close to equal in size.

Example 11. Fragments of Prior campaign c_{p1} are roughly equal in size:

$$(c_{p1}, [5, 4, 5])$$

One fragment of Prior campaign c_{p2} dominates the others in size:

$$(c_{p2}, [8, 1, 1, 2])$$

The important thing about $fragments\_by\_deviation$ is that it allows us to ignore cases where one fragment is dominant.

5.7.2 Merging rate

Merging is the assignments of two strings that were generated from different prior campaigns to one campaign.

Let P be a set of prior campaigns, S a set of strings, $p(s)$ a prior campaigns from which the string $s \in S$ was generated, and C a set of campaigns.

We define $constructs(p_c, c, S)$, which tells us if strings generated from a prior campaign $p_c \in P$ were classified into campaign $c \in C$ (were used to construct it):

$$constructs(p_c, c, S) = \exists s \in S : p(s) = p_c \wedge classify(s) = c$$

Prior campaign $p_c1, p_c2 \in P$ have been (partially) *merged* into campaign $c \in C$ in respect to a set of strings S if $merged(p_c1, p_c2, c, S)$ is true:

$$merged(p_c1, p_c2, c, S) = constructs(p_c1, c, S) \wedge constructs(p_c2, c, S)$$

How many prior campaigns were merged to create a campaign c can be calculated as follows:

$$merging\_rate(c, P, S) = |\{p_c : p_c \in P \wedge constructs(p_c, c, S)\}|$$

Campaigns with merging rate higher than one are of interest to us, because they could represent either a successful merging of fragmented campaigns, or, in case when we trust set P to be true (meaning that it is the set of campaigns from which strings in S were truly generated), an excessive merging.

Chapter 6

Optimizing the Algorithm

In the previous chapter, we described an algorithm for classifying spam SMS messages into campaigns. The algorithm depended on several hyperparameters. We described how they affect the classification algorithm, and alluded to what form they can take. In this chapter, we will propose various values of these hyperparameters, to produce what we will call an *instance* of the algorithm, which is a complete program that can classify a stream of strings into campaigns. We will then discuss what challenges the classification algorithm must overcome to classify messages into campaigns correctly.

6.1 Implementation

While the goal of this work is not to create software - but to design, test, and prototype an algorithm - an implementation of the designed algorithm is necessary to demonstrate it's usability on real data. `Python3` (version 3.8) was chosen for implementation, as it allows for rapid development and prototyping, supports both object oriented and functional programming, and speed of execution wasn't critical [1]. The implementation depends on the `NumPy` package [15], but only for ease of implementing matrix operations - it is not used to speed up the program, in fact it might be detrimental to performance. The program consists of several modules:

- `sequence` - Implements the concepts of token, token alphabet, token category, tokenization and joining sequences. Sequences themselves are implemented as python lists.
- `alignment` - Implements the Smith-Waterman and Needleman-Wunsch algorithms.
- `classify` - Implements the classify and streamClassify algorithms.
- `testbench` - Implements a command line tool for testing the streamClassify algorithm on a sample of messages and computing metrics.

Another script is necessary for the program to function, which is the implementation of hyperparameters of the classification algorithm. The `testbench` loads this script as a module, reads and samples a set of campaigns with LSH hashes (representing the prior campaigns) from a provided file, runs the streamClassify algorithm, computes metrics described in Section 5.7 and outputs four files (where `output` is the output name):

[1] When developing this solution for practice, speed will be naturally of utmost importance

- `output.log` - contains general information about the test, such as input filename, hyperparameters, number of samples, and aggregate metrics, such as fragmentation mean.
- `ouptut_campaigns.csv` - contains templates and campaign keys of all created campaigns.
- `output_campaigns_by_merging_rate.csv` - contains the top x campaigns sorted by merging rate
- `output_fragments_by_size.csv` - contains the top x fragmented prior campaigns sorted by fragment size (how many campaigns were messages from this prior campaign classified to)
- `output_fragments_by_deviation.csv` - contains the top x fragmented prior campaigns sorted by fragment deviation (the most evenly balanced fragments)

6.2 Naive Approach

In this section, we will consider a simple instance (values of hyperparameters) of algorithm *classify*, to demonstrate why a more sophisticated approach will be necessary.

The simplest possible parameters were chosen. Tokens are delimited by sequences of whitespaces.

```
$\Sigma_T$ := {(r'.*', 0)}
split := $\lambda$ str : split str on whitespace characters
Tokenize' := $\lambda$ str : tokenize(split, $\Sigma_T$, str)
```

Function 6.1: Naive Tokenize

Substitutions are scored by a strict match/mismatch strategy.

```
Score := $\lambda$ t1, t2 :
    if match(t1, t2):
        return matchReward
    else
        return -mismatchPenalty
```

Function 6.2: Naive Score

Alignment is performed using the Smith-Waterman alignment algorithm. Alignments are collapsed by performing a union of tokens and counting how many tokens were merged: if the number of alternatives is higher than a threshold, the token is marked as random.

```
Collapse = $\lambda$ $A_s$, $A_c$ : CollapseAlignment($A_s$, $A_c$, maxAlternatives)
```

Function 6.3: Naive Collapse

Alignment is deemed acceptable, if it's score is higher than a threshold and it's length is higher than a threshold.

```
Judge := $\lambda$ align, score:
    if score $\geq$ minScore $\wedge$ len(align) $\geq$ minLen:
        return True
    else:
            return False
```

Function 6.4: Naive Judge

Note that there is a number of named values that are not specified. Those are called *hyperparameters of second degree*, and they are discussed in more detail in Section 6.4. In Section 6.3 we will discuss what challenges the algorithm needs to overcome in order to successfully classify messages. We will do so by fixing those hyperparameters and analyzing the result of the algorithm.

6.3 Challenges

This section describes challenges that need to be overcome to transform the naive approach described above into a successful classification algorithm.

6.3.1 Universal Campaign

The universal campaign is the campaign $c_u = (id, [((.*, 0), .*, -1)])$. Any string can be said to belong to this campaign - this campaign will match any string. Once campaign c_u is present in a database of campaigns that we are learning, any new strings are automatically assigned to it, without changing its template. This behavior is an example of over-generalization and it will cause excessive merging(see Section 4.5.1), as all prior spam campaigns, however unrelated, will be classified into c_u.

The emergence of a universal campaign is caused by the alignment judging and scoring functions. The $classify$ algorithm will inevitably produce an alignment with a random token - this behavior is expected of it for reasons explained in Section 4.4 and Section 5.2.1. Let c_r be a campaign with a template containing a single random token. Under the match/mismatch scoring strategy, any string that is aligned against c_r will have a score at least equal to $matchReward$. If the minimal score of an alignment is lower than or equal to the match reward, c_r will inevitably transform into c_u.

Table 6.1 shows the creation of an universal campaign. All prior campaigns have been merged into a single campaign.

Table 6.1: creation of a universal campaign

(a) Parameters

matchReward	1
mismatchPenalty	1
gapOpenPenalty	1
gapExtendPenalty	1
maxAlternatives	1
minScore	1
minLen	1

(b) Metrics

prior campaigns	50
created campaigns	1
fragmentation mean	1.0
merging mean	50.0

A seemingly simple solution is to increase the minimal length of the alignment. This will, however, only delay the problem. Using the process described above to create a universal campaign with a single random token, universal campaigns with templates of any length can be constructed. This in turn makes increasing minimal alignment score an insufficient solution, as a universal campaign c_u^n with n random tokens, will always align with a score of *matchReward* $\cdot$ n, which is the highest possible score alignments of minimal length can achieve. Increasing the required minimal alignment score above *matchReward* $\cdot$ n would make alignments of minimal length impossible. Adjusting the judging function to accept alignments based on their score and length dynamically in respect to the length of the sequences being classified (as discussed in Section 6.4) likewise only delays the problem. Table 6.3 shows metrics collected from a test with such a judging function. Notice that the merging deviation is quite high. This is caused by the creation of an universal campaign.

```
def judgeAlignment(align:list, score:float, s:"list[Token]", c:"list[Token]")->bool:
    if (len(align) >= len(s)*minLenFrac and len(align) >= len(c)*minLenFrac) and score >= len(s)*minScoreFrac:
        return True
    else:
        return False
```

Function 6.5: Dynamic Length and Score Thresholds

Table 6.2: Creation of universal campaign with dynamic length and score thresholds

(a) Parameters

matchReward	1
mismatchPenalty	1
gapOpenPenalty	1
gapExtendPenalty	1
maxAlternatives	1
minScoreFrac	0.25
minLenFrac	0.5

(b) Metrics

prior campaigns	50
created campaigns	31
fragmentation mean	1.14
fragmentation deviation	0.40
merging mean	1.84
merging deviation	3.06

Universal campaigns are particularly troublesome, because, as explained above, they will always align with the highest score for an alignment of a given length.

We will now propose three methods for dealing with universal campaigns: limiting the number of random tokens in a template, discouraging the creation of random tokens, and retaining information about random tokens.

The simplest approach is to *limit the number of random tokens in a template.* To do this, we simply update the judging function to count the number of random tokens in an alignment and the alignment if the count is higher than some threshold.

```
--Judging
minScore := 1
minLen := 1
maxRandomTokens := 1
Judge := λ align, score:
        if score ≥ minScore ∧ len(align) ≥ minLen ∧ countRandomTokens(align) ≤ maxRandomTokens:
                return True
        else:
                return False
```

Function 6.6: Judging function limiting the number of random tokens

This approach introduces a new hyperparameter to optimize: maximum number of random tokens in a template. It is clear, however, that it must be strictly lower than the minimum alignment length to have any effect.

A more complicated approach is to *discourage the creation of random tokens* during alignment. To do this, we update the scoring function to detect when we are about to create a random token, and assign such substitution a lower score.

```
--Scoring
matchReward := 1
mismatchPenalty := 0
maxAlternatives := 1
randomTokenPenalty := 1
Score := λ t1, t2 :
        if t1 = t2:
                return matchReward
        else:
                if countTotalAlternatives(t1,t2) > maxAlternatives:
                        return -mismatchPenalty-randomTokenPenalty
                else:
                        return -mismatchPenalty
```

Function 6.7: Scoring function discouraging the creation of random tokens

This approach introduces a new hyperparameter to optimize: the random token creation penalty. The problem with this approach is, that while it does discourage the creation of universal campaigns, it also discourages the creation of random tokens, which is something that the algorithm is supposed to do, to a certain degree - we expect some spots in most campaigns' templates to be highly variable. A slight modification to this approach that would satisfy both the need to create random tokens to describe high variability and to discourage the creation of universal campaigns is be to discourage the creation of *too many* random tokens. As discussed in Section 4.4, most campaigns will consist of more fixed tokens than random. The penalty for creating a random token might start small, and gradually increase with each random token created.

The third approach is to gather information about random tokens. This can be done by employing *token categories*. When we referred to a *random token* above, we meant token $t_r = ((.*, 0), .*, -1)$, which matches any string. However, in Section 2.3 we defined random token to be any token with alternative count equal to -1. Therefore, random tokens can have any category. By using a token alphabet with categories that don't accept all strings, we can avoid creating universal campaigns by removing their prerequisite: an universal token.

Consider the following strings as an example of a spam campaign:

G−244763 is your Google verification code.
720621 is your Skrill authentication code.
427612 is your NETELLER authentication code.

We can see that the first word in the template of this campaign (the verification code) is highly variable, and will likely become a random token. If we use the trivial alphabet $\Sigma_{T_0} = \{('.*', 0)\}$, we run a risk of learning a universal campaign. Instead, we can use a slightly more complicated alphabet $\Sigma_{T_1} = \{('(\backslash w*)?d+', 0)), ('\backslash w+', 1))\}$. Now, even though the generated template starts with a random token, it will not accept all string - only ones that start with a verification code.

To make this approach work, we will need to adjust tokenization. The precise form the used token alphabet will take depends solely on the data being classified, however, the union of patterns of all categories in the alphabet should be a superset of the language of the data, otherwise, the algorithm will be unable to tokenize some strings. The problem of a universal campaign has already been solved by this, as random tokens will no longer match everything, however, the use of categories allows us to improve the algorithm further, mainly by using categories in scoring substitutions.

6.3.2 Low Scoring Short Message

The score of an alignment is the sum of all substitution scores plus the sum of all gap scores. Consider an alignment of length n. The highest possible score for this alignment is $max(score) \times n$, where $max(score)$ is the maximum possible substitution score. Therefore, the highest score a short messages can achieve is lower than that of longer messages. This presents a problem, because it means that if we expect messages of different lengths, short messages might never reach our fixed score threshold, even when aligned against themselves. However, setting the score too low will make it useless.

Note that this does not have to do with the quality of the alignment, in respect to the ratio of matching and mismatching tokens. Consider the following instance of the naive algorithm:

Table 6.3: Creation of universal campaign with dynamic length and score thresholds

(a) Parameters

matchReward	1
mismatchPenalty	1
gapOpenPenalty	1
gapExtendPenalty	1
maxAlternatives	1
minScore	3
minLen	1

(b) Metrics

prior campaigns	50
created campaigns	48
fragmentation mean	1.14
fragmentation deviation	2.69
merging mean	1.48
merging deviation	0.85

Note the high value of fragmentation deviation, that is caused by messages from a prior campaign with a short template being unable to merge, due to the high score threshold.

Example 12. Template: Telegram code <NUMBER>
Messages in sample: 20
Number of campaigns: 20

The highest alignment score this messages from this campaign could have achieved is 1 (two fixed tokens(1+1), one variable(-1)), which is lower than the threshold, and therefore even the best alignment possible was not judged to be an acceptable template.

6.3.3 Forgotten Extremes

1 Dear New Rail Sheba User, You are opening a new account at Rail Sheba App. Please activate your user account using this OTP: 214423 Thank you

2 Dear User, Please activate your user account using this OTP: 624080 Thank you.

These messages belong to different prior campaigns. However, the alignment algorithm might align them with a sufficient score to be merged into the same campaign by cutting of their heads.

Template: `Please activate your user account using this OTP: <.*> Thank you.`

The same issue might arise when two messages have different tails, or even when the matching part is in the middle.

A slightly less problematic instance of this problem is when extremes are forgotten when classifying messages that do belong to the same campaign. Consider:

```
1        Dear Client, TK. 20776.00 credited to ∗∗∗82520 by Transfer dated 2020−09−08
             16:16. Remaining balance 177804.08. MTB Helpline ∗∗∗∗9
2        Dear Client, TK. 100000.00 credited to ∗∗∗98648 by On−Line Cash dated
             2020−09−08 3:58. Remaining balance 3016218.73. MTB Helpline ∗∗∗∗9
```

Template: `dated 2020-09-08 <.*> Remaining balance <.*> MTB Helpline ****9`

The alignment algorithm decided that the head of the messages is too variable to be worth hassling with, even though there are some exact matches.

These are examples of over-generalization, and are caused by the local nature of the Smith-Waterman alignment algorithm. One way to fix this problem is to use a global alignment algorithm, like Needleman-Wunsch. That way, the entire message is considered, and differences in lengths are correctly classified as gaps.

6.3.4 Rejecting Suboptimal Alignments

The alignment algorithm in our algorithm returns a single alignment: the optimal one. However, the judging function that decides whether this alignment is suitable for a template might depend on more metrics than just the alignment score. Therefore, it is possible that the optimal alignment might not pass the judging process, while a sub-optimal alignment might.

One solution to this problem is to return all alignments and then loop over them, until we find one that passes the judging function, or we run out of alignments. This, however, raises the time complexity of our algorithm considerably. Firstly, while the smith-waterman algorithm can be optimized to run in $O(n)$ time (where n is the length of the shorter sequence) for situations when only the optimal alignment is of interest to us, it will run in $O(nm)$ time when we want all alignments. Secondly, the rest of the algorithm will slow down as well. The alignment joining and alignment judging will have to run x times in worst case (where x is the number of alignments).

6.3.5 Good enough match

The *streamClassify* algorithm compares messages with campaigns sequentially, but it does not specify what order the campaigns are in. This might result in a situation where a message is classified into a campaign with which it aligns sub-optimally.

One solution to this problem would be to compare the message with all camapigns, then select the optimal match. However, this would only be feasible if the campaign space was small enough.

6.4 Hyperparameters of Second Degree

In chapter 5 we described the hyperparameters of the classification algorithm. In Section 6.2, we fixed those parameters to obtain an *instance* of the algorithm. However, to do so, we used named constants such as `minScore` or `minLen`. These constants are the hyperparameters of this particular *instance* - the algorithm will behave differently when we change their values. We refer to these as to *hyperparameters of second degree*, and they can be different for different instances of the classification algorithm.

6.4.1 Substitution Score

In biology, substitution score of two proteins represents the likelihood they are to mutate into one another. In our algorithm, the substitution score represents the likelihood of two tokens being generated from the same field in a template. We say that a score for two matching strings (two tokens that were likely generated from a fixed field in a template) is 1, and two tokens that do not match have score of -1. This means, that an alignment with score 0 has an equal number of matching and mismatching tokens.

6.4.2 Minimal Alignment Score

The minimal alignment score an alignment must achieve to be considered acceptable depends on what positive score is assigned for. In the algorithm instance proposed in Section 6.2, score increased by 1 for each matching token, and decreased by an equal amount for each mismatching token and each unit of a gap. The alignment score, therefore, represents the amount of matching tokens compared to the the amount of mismatching tokens and gaps. The minimal alignment score represents how many more matches than mismatches and gaps are required in an alignment to be considered a template.

The alignment score is not unbound. The highest scoring possible alignment would be an alignment of two identical sequences of length $n \in \mathbb{N}$, and this score would be n. Conversely, the lowest score an alignment could achieve is 0 - this is due to the nature of the Smith-Waterman alignment algorithm. An alignment with such score would be optimal (i.e. returned by the alignment algorithm) only if the two sequences being aligned didn't share a single matching token, regardless of their lengths.

Let n be the length of the shortest possible sequence that could be constructed from all messages being classified. If the minimal alignment score is a fixed number $m \in \mathbb{N}$ it must be lower than n, otherwise, some messages will be unable to form campaigns at all. This problem can be mitigated by making mismatches lowering the alignment score (as opposed to just not increasing it). This allows us to keep the minimal score close to zero; if a match increases the score, and a mismatch decreases it by the same amount, then a score of zero means the same number of matches and mismatches are present in the alignment.

The actual value depends on the data being classified, and has to be chosen through experiment.

6.4.3 Minimal Alignment Length

The minimal length an alignment must achieve to be considered acceptable as a template is a parameter that is only applicable if the alignment method used is local - when aligning sequences globally, the alignment will always have the length of the longer sequence.

Similar to minimal score, if the minimal length were to be a fixed number, it would need to be lower then the length of the shortest message. However, unlike the minimal score, which only represents the required ratio of matches and mismatches and isn't actually connected to the length of the message (and it is acceptable to set it to a low fixed number, as explained above), we expect messages of different lengths to appear in the data being classified. The minimal length should therefore be set dynamically, in respect of the lengths of the messages being aligned.

The first explored approach is to set the minimal length to an average of the two sequences being aligned. This approach is simple, and eliminates a optimizable hyperparameter, however the change of length a campaigns template undertakes is unpredictable, and it is influenced by the length of a message that might not even belong to that campaign. The second approach is to instead set a number of tokens that a campaign can lose. This produces a more predictable change in the length of the template. Unfortunately, neither of these approaches solves the problem of a campaign continuously shrinking, until it is only one token long. To solve this, a length change limit can be introduced: a campaign can not become shorter than its starting length minus this number.

The maximum token loss exposes a problem in our approach to classification. Quite obviously, we do not want to lose any tokens, because that means loosing information about the campaign's generation template. This encourages excessive merging. We would much rather the campaign gained random or optional tokens.

6.5 Improved Classification Algorithm

Drawing on the insights presented in Section 6.3 and Section 6.4, we improve the classification algorithm. The simple tokenization used in Section 6.2 performed well enough, however, it failed in certain situations, for example when the message lacked a space between what we would consider tokens.

```
Your A/C No:5637*****0631 has been CREDITED Tk.24,432.00 on 08−SEP−2020
    by Transfer. Current Balance Tk.26900. Thank You, R****i Bank Ltd.
```

We would like to separate the word "No:„ from the account number, or the currency symbol "Tk.„ from the amount, so we can recognize and treat them as numbers. To do so, we switched from splitting the message by a pattern, to extracting substrings that match a pattern from the message. The precise form this pattern takes depends on the data being analyzed. For this experiment, we chose to extract time (not that many formats), dates (only a subset of formats that appears in the testing data, as there are too many possible formats), emails, numbers, words with internal dashes, dots, hyphens, and apostrophes, and a subset of special symbols. We also decided to throw away punctuation marks, as these were often missing (increase in noise) and didn't contribute to the alignment significantly, as we analyze neither the semantics nor the syntax of the messages.

```
#Token Alphabet (the third element creates the tree structure: points
    to a parent)
Σ_T := {
       wordCategory := ('(?P<word>\w+(?:[-.\\/\']\w+)*'?)', 0, None),
       dateCategory :=
       ('([0-3]?[0-9]
       (?P<date_sep1>[/.-])
       (?:[0-3]?[0-9]|Jan|Feb|Mar|Apr|Aug|Sep|Oct|Nov|Dec)
       (?P=date_sep1)
       (?:[0-9]{2})?[0-9][0-9]
       |(?:[0-9]{2})?[0-9][0-9](?P<date_sep2>[/.-])
       (?:[0-3]?[0-9]|Jan|Feb|Mar|Apr|Aug|Sep|Oct|Nov|Dec)
       (?P=date_sep2)
       [0-3]?[0-9])', 1, wordCategory),
            timeCategory := ('(((?:2[0-3])|(?:[01]?[0-9])):([0-5]?[0-9])
                (?::[0-5]?[0-9])?)', 1, wordCategory),
            emailCategory := ('(?P<email>\S+@\S+)', 1, wordCategory),
            numberCategory := ('(?P<number>[\d*]+(?:[,-][\d*]+)*(?:\.[\d*]+)
                ?)', 1, wordCategory),
            specialCategory := ('(?P<special>[^\w\d\s,.?!]+)', 1,
                wordCategory),
    }

    #tokenize
    split := λ
    #pattern is a regular expression created by performing a union of all
        patterns of all categories in $\Sigma_T$
    pattern := reduce(λ x, y : x.pattern + '|' + y.pattern, $\Sigma_T$)
    split = λ s~: find all nonoverlaping substrings of s~that match pattern
    Tokenize = lambda s~: tokenize(split, alphabet, s)
```

Function 6.8: Improved Tokenization

We categorized tokens into categories roughly corresponding to the tokenization groups. Certain types of tokens are more likely to have been generated from highly variable fields in the template than others: numbers, dates, and times. We consider tokens in these categories to “always match„ - we don’t penalize them for their values not matching. We also consider tokens with different categories to be highly unlikely to align, so we penalize them more.

```
matchScore = 1
mismatchPenalty = 1
categoryMismatchPenalty = 2
highlyVariableCategories = {dateCategory, timeCategory, numberCategory}

Score = λ t1, t2:
        if t1[0] == t2[0]: #categories match
                if t1[0] ∈ highlyVariableCategories:
                        return matchScore
                else:
                        if t2.match(t1):
                                return matchScore
                        else:
                                return -mismatchPenalty
        else:
                return -categoryMismatchPenalty
```

Function 6.9: Improved Score

We collapse the alignment in the same way we did in 6.2. We judge an alignment acceptable, if it has a score higher than 0, or in other words, the number of matches is greater than the number of mismatches and gaps. This constant worked well in testing.

```
#collapse
maxAlternatives=5
Collapse = λ $A_s$, $A_c$ : collapseAlignment($A_s$, $A_c$, maxAlternatives)

#judge
minScore = 0
```

```
        Judge = λ score : return score >= minScore
```

Function 6.10: Improved Collapse and Judge

Most importantly, we abandon Smith-Waterman algorithm in favor of Needleman-Wunsch. This change makes the algorithm significantly better. Local alignment is useful when we are interested in finding similar substrings in very long strings, while ignoring regions that are very different. However, we are interested in these regions, as they provide valuable information, namely the fact that the compared strings might have a similar (local) part, but overall (globally), they are very different. In other words, Smith-Waterman is lossy in a way that we can not afford.

We also implement some simple ways to combat noise. Consider these two messages:

```
Your Apple ID Verification Code is: 255253
Your Fiverr verification code is: 7895
```

They obviously belong to the same campaign. However, the algorithm described in Section 6.2 would fragment this campaign, because of the subtle difference in word capitalization. We can implement a very simple fix to this problem: convert all text to lowercase characters during tokenization.

Table 6.4 shows how the improved algorithm performs on smaller subsets (that are easy to analyze by hand). This behavior seems to be consistent over different samples of similar size. Table 6.5 shows how the algorithm performs on a large dataset, namely the one described in Section 4.5.2. The performance is consistent with the smaller tests. The increase in merging rate is due to the fact, that some campaigns in the dataset are more fragmented than others by LSH.

Table 6.4: Improved Algorithm Performance on a Small Sample

(a) Parameters

matchReward	1
mismatchPenalty	1
categoryMismatchPenalty	2
gapOpenPenalty	1
gapExtendPenalty	1
maxAlternatives	5
minScore	0

(b) Metrics

sample size	1000
prior campaigns	50
created campaigns	34
fragmentation mean	1.06
fragmentation deviation	0.24
merging mean	1.56
merging deviation	0.82

Table 6.5: Improved Algorithm Performance on a Large Sample

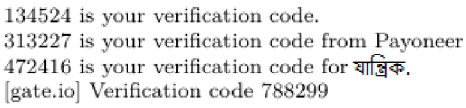
(a) Parameters

matchReward	1
mismatchPenalty	1
categoryMismatchPenalty	2
gapOpenPenalty	1
gapExtendPenalty	1
maxAlternatives	5
minScore	0

(b) Metrics

sample size	58450
prior campaigns	267
created campaigns	83
fragmentation mean	1.06
fragmentation deviation	0.29
merging mean	3.4
merging deviation	13.9

Some prior campaigns still get fragmented. Consider the following messages: [2]



```
1     134524 is your verification code.
2     313227 is your verification code from Payoneer
3     472416 is your verification code for যান্ত্রিক.
4     [gate.io] Verification code 788299
```

Messages 1 and 2 are correctly classified into the same campaign (they actually get correctly merged in respect to LSH clustering, see lower). Message 3 is incorrectly separated into it's own campaign. This is caused by the tokenization mishandling bengali alphabet and incorrectly splitting the last word of this message into 9 tokens. Message 4 is an example of correct fragmentation; our algorithm corrected excessive merging of the LSH algorithm used to classify this dataset. Message 4 was classified into the same prior campaign by the LSH algorithm as messages 1 and 2, even though we can see they were not generated from the same template.

The merging rate of the algorithm remains high, however, not all the merges are incorrect. Consider the following messages:

```
1     Dear Client, TK. 90000.00 debited from ∗∗∗44957 by Cash dated 2020−09−08 3:51.
          Remaining balance 1811.00. MTB Helpline ∗∗∗∗9
2     Dear Client, TK. 50.00 debited from ∗∗∗88227 by Transfer dated 2020−09−08 4:11.
          Remaining balance 29818.00. MTB Helpline ∗∗∗∗9
3     Dear Client, TK. 18.21 credited to ∗∗∗00386 by Transfer dated 2020−09−08 16:16.
          Remaining balance 1244.43. MTB Helpline ∗∗∗∗9
```

These messages were clearly generated from the same template. However, the LSH algorithm classified them into three different campaigns: it fragmented the campaign. Our algorithm correctly classified all three messages into the same campaign: it corrected the LSH fragmentation problem.

6.5.1 Edit Distance Scoring

In Section 5.4 we discussed the use of edit distance for substitution scoring as a means to deal with noise in the messages. However, in the course of testing, this approach turned out to be ill suited for template extraction. It provided no noticeable improvement, while decreasing the classification speed significantly. This is caused by the fact that data on which this algorithm was evaluated does not contain random noise in the form of extra letters interspersed in the message.

6.6 Generated Tempaltes

An important thing to note is that the templates our algorithm generates are very different than the templates we describe in Section 4.4. Consider the following messages:

```
1     Dear Client, TK. 50.00 debited from ∗∗∗88227 by Transfer dated 2020−09−08 4:11.
2     Dear Client, TK. 90000.00 credited from ∗∗∗44957 by Online Cash dated
          2020−09−08 3:51.
```

[2] At this point an image is included instead of text, as LaTeX, not unlike our program, can't handle the bengali script

The template that might generate these messages might look something like this:

```
Dear Client, TK. {AMOUNT} {"debited"|"credited"} from {ACCOUNT} by
    {"Transfer"|"Online Cash"} date {DATE} {TIME}.
```

However, the template our algorithm generates looks like this:

```
[(dear), (client), (tk), (.∗, NUMBER), (debited|credited), (from), (.∗, NUMBER),
    (by), (online|ϵ), (cash|transfer), (dated), (.∗, DATE), (.∗, TIME), (remaining)]
```

For one, the template loses certain characters, such as a comma after client, and even the text that was converted into tokens is transformed during tokenization - in this case, it was converted to lowercase. More importantly, however, notice how the algorithm handles the `{"transfer"|"online cash"}` field. While the prior template contains this as a single field, that can generate a variable amount of words, our algorithm is forced to treat one of the words as optional (a gap in the alignment), and the other as variable (a mutation).

6.7 Merging Campaigns

The algorithm is designed in such a way, that it can be used for aligning two campaigns. This can be useful when we already have messages grouped into campaigns by some other method, for exmaple LSH, but we are worried that they might be fragmented. The classification algorithm can be used as a second layer over the LSH method, to merged possible fragments.

This attribute can also be useful for reducing templates created in a distributed manner, for example some kind of MapReduce model [9].

Chapter 7

Conclusion

While email is still the most popular medium for sending electronic spam, it's popularity has been waning due to advances in combating it. Both spammers and legitimate companies are taking more and more interest in sending bulk messages through SMS. In this paper, we designed an algorithm for performing one essential task in spam detection: grouping messages into campaigns.

We made a brief summary of what spam is, focusing on SMS communication. We described what is a campaign and how messages from a campaign are generated using a template.

We designed and implemented an algorithm for classifying messages into campaigns, while simultaneously learning the generation templates of those campaigns. We based the algorithm on aligning two sequences of words, as opposed to sequences of letters. The idea of alignment was adopted from bioinformatics, where it is used to find similar regions in protein or DNA strings. We focused on adapting the Smith-Waterman algorithm for local alignment, but discovered that local alignment is ill suited for this task. We hypothesized that global alignment is a better suited for our purposes, and confirmed this hypotheses by replacing Smith-Waterman with Needleman-Wunsch alignment algorithm. The algorithm was designed to work iteratively, incrementally generalizing the campaign templates it was learning. The more time consuming operations are only performed when new messages do not match the template already. We confirmed that the final product performs acceptably well on the data provided, even correcting fragmentation errors that the older LSH method made. However, we also learned that parameters will have to be tweaked by hand depending on the use case and the data being classified.

Bibliography

[1] *3GPP TS 23.038, Alphabets and language-specific information.*

[2] *3GPP TS 23.040, Technical realization of the Short Message Service.*

[3] *3GPP TS 29.002, Mobile Application Part specification.*

[4] *Algorithms for Molecular Biology.* Archived from the original on 2013-06-26. Jan. 1, 2006. URL: https://web.archive.org/web/20130626060959/http://www.biogem.org/downloads/notes/Gap%20Penalty.pdf (visited on 04/11/2021).

[5] J. Bergstra, D. Yamins, and D. D. Cox. "Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures." In: *Proc. of the 30th International Conference on Machine Learning (ICML 2013).* 2013.

[6] *BLAST: Basic Local Alignment Search Tool.* URL: https://blast.ncbi.nlm.nih.gov/ (visited on 04/21/2021).

[7] *BLOSUM62 scoring matrix for amino acid substitutions.* Mar. 6, 2021. URL: https://bio.libretexts.org/@go/page/17548 (visited on 04/21/2021).

[8] Cloudmark. "SMS Spam Overview — Preserving the value of SMS texting". In: (). URL: https://www.cloudmark.com/en/resources/white-papers/sms-spam-overview-preserving-value-sms-texting (visited on 07/04/2021).

[9] J. Dean and Ghemawat S. "MapReduce:Simplified Data Processing on Large Clusters". In: (2004). URL: https://static.googleusercontent.com/media/research.google.com/es/us/archive/mapreduce-osdi04.pdf (visited on 07/05/2021).

[10] Sarah Jane Delany, Mark Buckley, and Derek Greene. "SMS spam filtering: Methods and data". In: *Expert Systems with Applications* 39.10 (2012), pp. 9899–9908. ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2012.02.053. URL: https://www.sciencedirect.com/science/article/pii/S0957417412002977.

[11] H. Ebbinhaus, J. Flum, and W. Thomas. *Mathematical Logic.* 2nd ed. Springer-Verlag New York, 1994. ISBN: 978-1-4757-2355-7.

[12] *FASTA Sequence Comparison.* URL: https://fasta.bioch.virginia.edu (visited on 04/21/2021).

[13] *Glossary.* URL: http://rosalind.info/glossary/gap-penalty/ (visited on 04/11/2021).

[14] *Glossary.* URL: http://rosalind.info/glossary/ (visited on 04/11/2021).

[15] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[16] C. de la Higuera. *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, 2010. ISBN: 9781139486682. URL: https://books.google.cz/books?id=XAOE5V9B4dUC.

[17] Piotr. Indyk and Rajeev. Motwani. "Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality". In: *STOC '98: Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*. Dallas, Texas, USA: Association for Computing Machinery, 1998. ISBN: 0897919629.

[18] C. Kreibich and C. Kanich. *Spamcraft: An Inside Look At Spam Campaign Orchestration*. URL: https://www.usenix.org/legacy/event/leet09/tech/full_papers/kreibich/kreibich_html/index.html (visited on 06/19/2021).

[19] William J. Masek and Michael S. Paterson. "A faster algorithm computing string edit distances". In: *Journal of Computer and System Sciences* 20.1 (1980), pp. 18–31. ISSN: 0022-0000. DOI: https://doi.org/10.1016/0022-0000(80)90002-1. URL: https://www.sciencedirect.com/science/article/pii/0022000080900021.

[20] Saul B. Needleman and Christian D. Wunsch. "A general method applicable to the search for similarities in the amino acid sequence of two proteins". In: *Journal of Molecular Biology* 48.3 (1970), pp. 443–453. ISSN: 0022-2836. DOI: https://doi.org/10.1016/0022-2836(70)90057-4. URL: http://www.sciencedirect.com/science/article/pii/0022283670900574.

[21] OECD. "Background Paper for the OECD Workshop on Spam". In: 78 (2004). DOI: https://doi.org/https://doi.org/10.1787/232784860063. URL: https://www.oecd-ilibrary.org/content/paper/232784860063.

[22] The Spamhaus Project. *The Definition of Spam*. URL: https://www.spamhaus.org/consumer/definition/ (visited on 05/19/2021).

[23] The Spamhaus Project. *The World's Worst Spam Enabling Countries*. URL: https://www.spamhaus.org/statistics/countries/ (visited on 05/20/2021).

[24] T. F. Smith and M. S. Waterman. *Identification of Common Molecular Subsequences*. 1981.