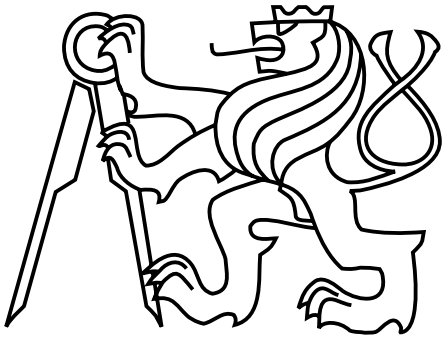
Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Cybernetics



Bachelor's Thesis

Isolation of Business Logic Represented by ETL Processes by Machine Learning Algorithms

Juraj Žilt

Supervisors: Mgr. Petr Hála, Ing. Ondřej Kuželka, Ph.D.

Study Programme: Open Informatics

Field of Study: Artificial Intelligence and Computer Science

May, 2023

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Žilt Juraj** Personal ID number: **498867**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Cybernetics**

Study program: **Open Informatics**

Specialisation: **Artificial Intelligence and Computer Science**

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Isolation of Business Logic Represented by ETL Processes by Machine Learning Algorithms

Bachelor’s thesis title in Czech:

Izolace byznysové logiky reprezentované pomocí ETL procesů za pomoci strojového učení

Guidelines:

The thesis aims to design a method for SQL snippet classification based on machine learning. The system should be able to group snippets with identical or similar usage. The final product should detect business logic in SQL scripts, which can then be used in practice for impact anlysis, business revision and optimizations.
1.) Understand the leading implementations of similar problems and their solutions.
2.) Transform the SQL codes into data in a form suitable for further analysis.
3.) Detect typical structures of provided files and thier funcitonalities using existing algorithms, e.g. for code plagiarism.
4.) Quantify identifiers for logical structure analysis.
5.) Classify snippets based on code and logical structure using machine learning algortihms.
6.) Anlayze the results obtained by algortitm.

Bibliography / sources:

[1] Wise, Michael. (1993). String Similarity via Greedy String Tiling and Running Karp−Rabin Matching. Unpublished Basser Department of Computer Science Report.
[2] McInnes, Leland and Healy, John and Melville, James: UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction, arXiv, 2018
[3] Balder Cate and Victor Dalmau: The Product Homomorphism Problem and Applications, Dagstuhl Publishing, Germany,2015

Name and workplace of bachelor’s thesis supervisor:

Mgr. Petr Hála Profinit EU, s.r.o., Praha

Name and workplace of second bachelor’s thesis supervisor or consultant:

Ing. Ondřej Kuželka, Ph.D. Intelligent Data Analysis FEE

Date of bachelor’s thesis assignment: **21.01.2023** Deadline for bachelor thesis submission: **26.05.2023**

Assignment valid until: **22.09.2024**

Mgr. Petr Hála	prof. Ing. Tomáš Svoboda, Ph.D.	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature	Head of department’s signature	Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student's signature

Acknowledgements

Firstly, I want to thank Profinit, which came up with the topic, provided me with data, and trusted my abilities to fulfill the requirements. I am also grateful for Petr Hála and Petr Hájek, who regularly discussed possible solutions with me and patiently waited in moments when the progress was negligible. In the end, I am thankful for Ondřej Kuželka, my university solicitor, who took responsibility without knowing me and discussed all the problems with interest.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, 17.05.2023 Juraj Žilt

Abstrakt

Moderní společnost si v dnešní době může udržet svou existenci pouze díky kvalitním datům a jejich analýze. Data se obvykle sbírají a analyzují několik desetiletí. Bohužel spolu s daty na různých platformách roste i počet transformací Business intelligence. Když se všechna data chtějí přenést do cloudu, je třeba tyto transformace modernizovat. Cílem této bakalářské práce je vytvořit posloupnost algoritmů schopných rozpoznat úlohy Business intelligence.
Tato práce využívá techniky detekce plagiátů k odhalení technické struktury opakujících se zdrojových kódů. Nesupervizovaná technika Uniform Manifold Approximation and Projection pro redukci dimenze později analyzuje identifikované struktury. Výstup slouží jako vstup pro další nesupervizovnou metodu, Hierarchical Density-Based Spatial Clustering of Applications with Noise, která generuje přiřazení pro redukovanou dimenzi. Tyto přiřazení jsou považovány za Business intelligence a jsou interpretovány rozhodovacími stromy natrénovanými na neredukovaných datech.
Výsledky dokazují funkčnost zvolených algoritmů, které dokázaly odhalit příkazy ve zdrojových kódech SQL definující jednotlivé úlohy používané v celém množství souborů.

Klíčová slova: Business inteligence, Shoda sekvencí, Detekce logiky SQL

Abstract

A modern company nowadays can only keep its existence with good data and data analysis. The data is usually collected and analyzed over multiple decades. Unfortunately, the number of Business intelligence transformations grows with the data on various platforms. When all the data want to be transferred into the cloud, these transformations need to be modernized. This bachelor thesis aims to create a stream of algorithms able to recognize Business intelligence tasks.
This thesis uses plagiarism detection techniques to detect the technical structure of the repetitive source codes. The unsupervised learning technique Uniform Manifold Approximation and Projection for dimension reduction later analyzes the identified structures. The output serves as input for another unsupervised method, the Hierarchical Density-Based Spatial Clustering of Applications with Noise, which generates labels for the reduced dimension. These labels are considered Business intelligence and interpreted by decision trees trained on unreduced data.
The results prove the functionality of the chosen algorithms, which were able to detect statements in the SQL source codes defining the individual tasks used throughout the multiple files.

Keywords: Business inteligence, Matching sequances, SQL logic detection

Contents

List of Figures

Chapter 1

Introduction

Every company's operational goals are improving performance, efficiency, sales, product quality, customer satisfaction, and other similar indicators. Most companies would quickly perish without these ambitions, and therefore, a modern company cannot do without an analytical team and active data collection or procurement.

Middle to large-scale enterprises, whose businesses have been digitally transformed and are highly dependent on data processing, gather enormous loads of data for multifold purposes. However, not all collected information is necessary for analytical objectives. Therefore, in computer science, there exists a specific group of tasks called Business intelligence (BI), which almost exclusively supports the enterprise's analytical, planning, and decision-making activities by means of efficient data transformation.

In order to sustain the dynamics of today's world and grow their businesses, companies have to modernize their data processing. Unfortunately, these BI transformations have been implemented over the years, even decades, based on the current business needs and grown in size. It means BI transformations might be implemented on different technology platforms and written by numerous programmers under various methodologies. Thus, modernization is no longer a manually solvable task. Incoming programmers then have two options when starting to work for a company. The first is to reuse existing source code by calling prepared functions/scripts, and the second is to rewrite the same functionalities again. Since it is impossible to get to know all the existing code, and typically there is a lack of proper documentation, new programmers tend to stick to the second option. It causes the scripts to be even longer and more variate.

This work aims to automatically extract business logic from SQL source codes provided by Česká spořitelna, group them up, and assign them their role in the data transformation activities. Therefore, we will start with basic types of algorithms for detecting plagiarism, which we will see as a baseline[1] and end up with more complex algorithms that fall into the category of machine learning.

The rest of this thesis is organized as follows. In Chapter 2, the necessary background needed in this thesis is covered. Later in Chapter 3, the proposed solution is discussed, which is followed by Chapter 4 talking about the implementation of the used algorithms. Chapter

[1]The decision-making level against which we will evaluate the impact of the following more advanced algorithms.

5 describes the required steps to obtain the results. In the end, Chapters 6 and 7 talk about achieved results and possible improvements.

Chapter 2

Background

The following sections will build the knowledge necessary to understand this bachelor's thesis. They should also help better understand all possible impacts, such as error uncovering, documentation automatization, and mainly BI detection.

2.1 Business intelligence

Business intelligence (BI) represents a specific collection of tasks in computer science, which almost exclusively support the analytical, planning, and decision-making activities of companies and organizations [5]. BI applications do not create new data but use data created by transactional applications stored in source databases.[1] These data are transformed using ETL (Extract, Transform, Load)/"data pump", which can be thought of as a set of programs providing data extraction (Extract) from source databases, transformation (Transform) into other data structures, and subsequent storage (Load) into analytic databases. See Figure 2.1.

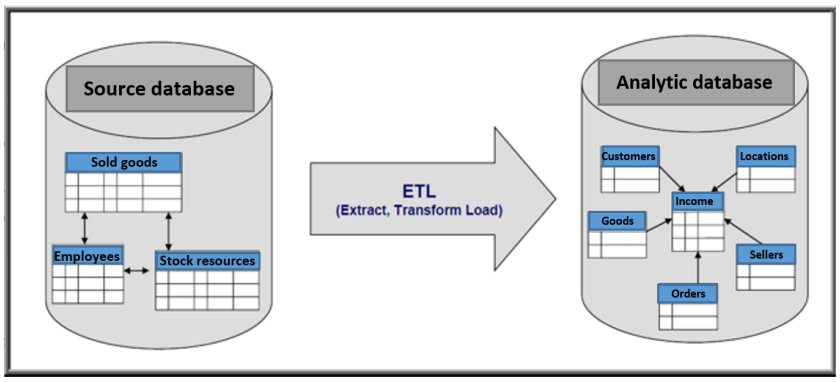


Figure 2.1: Data acquisition for analytical activities [5].

[1] "Source database" is not the technical name/type of the database, but only the naming of the database from the BI point of view.

Unlike source databases, analytic databases are optimized to provide analytical information efficiently. Fundamental differences between source and analytic databases are:

- Source databases are primarily designed to store newly created data and its updates.
- Source databases maintain data at the maximum level of detail, while an analytic database stores only data relevant to the analysis.
- Source databases hold data with actual dates. An analytic database uses the time dimension. Thus, data collections are stored in time frames.

2.2 Lexical analysis

A primary task of lexical analysis is to relate morphological variants to their lemma that lies in a lemma dictionary bundled up with its invariant semantic and syntactic information [7]. Thus, in the subsequent parsing, we do not distinguish between words from the set, e.g. bring, brought, bringing, etc.. However, we consider it always to be "bring," which, from an algorithmic point of view, will reduce the number of words found in the dictionary and improve the ability to quantify the actual occurrence of a word in the text. Otherwise, it is necessary to have extensive input data where the algorithm can find a sufficient number of appearances of every word variant.

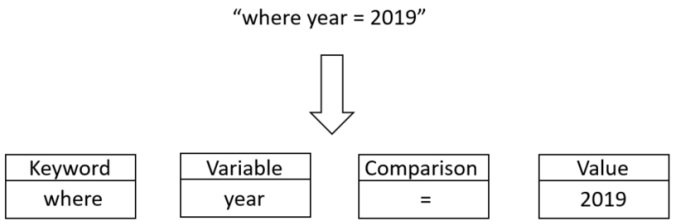


Figure 2.2: Tokenization of SQL source code.

Lexical analysis of source code is a transformation of input text into a much more compact token stream (keywords, parentheses, operators, identifiers, ...) [21]. In this thesis, tokens represent the same code structure, e.g., a token of type "Where", "Variable", ..., so we can create a "technical code structure." See Figure 2.2. The main benefit is that we do not distinguish between tokens of the same type, even when they contain different logic.[2] Such representation of the source code will help us detect duplicities later in Section 2.3.

2.3 Detection of matching sequences

Detecting matching sequences means identifying identical or similar patterns within a given dataset. Multiple algorithms can do it for numerous purposes, such as genetic research, finances, or data science. The most famous algorithms belonging to the group called "edit

[2]Currently there is no difference between CUSTOMER = 5 and AccountSTATUS = 1000.

distances"[3] in the programming community are, for example, the Levenshtein distance (LD) or the Longest common subsequence problem (LCS). Both of these algorithms are used for code similarity recognition. Unfortunately, they search for a single sequence in entire files and thus cannot recognize the transposition of code blocks, which is common practice in programming.

Other possible lesser-known algorithms are Knuth Morris Pratt (KMP), Boyer-Moore (BP) and Running-Karp-Rabin Greedy String Tiling (RKRGST). All three of them can detect transpositions and multiple duplicities within one file. Unfortunately, the first two algorithms are very memory intensive.

2.3.1 Levenshtein distance

Levenshtein's distance is a metric measuring the distance between two sequences. In the case of LD, the "distance" is the number of individual transforming operations required to modify sequence A to be identical to sequence B. A sequence is usually a chain of letters, words, numbers, or in our case, tokens carrying the information we seek for. In LD, we use three basic operations to transform text, namely *Substitution*, *Deletion* or *Insertion* [13]. Thus, LD searches for the minimum number of operations needed to transform the string A to B.

In case of file A with tokens CD and file B with tokens DC[4] were compared, LD would evaluate the distance as two because it would require deleting token D from file B and, later on, inserting D to the end of file B. It can mislead us into thinking that files A, and B are very different since the length of the files is just two, but in reality, from the programming point of view, other order of code blocks does not imply unique functionality.

There exists another disadvantage of LD for files of various lengths. If A had 100 tokens and B just 10, the distance is immediately 90 due to insert/delete operations. Therefore, the LD was not considered a practical approach for this thesis.

2.3.2 Karp-Rabin Algorithm

Karp-Rabin Algorithm (KRA) is an algorithm that searches for similarities based on matching hashes[5]. KRA starts by precomputing the hashes of all the sub-similarities (a predetermined minimum range of required similarity length) of file A, against which we want to compare file B. If A is a sequence of ten tokens and the specified similarity length is eight, we precompute three hashes (1-8, 2-9, 3-10). We then iterate over the tokens of B, which we hash using the same principle. When B generates a hash that exists in A, the minimal match has been found. Afterward, the following tokens[6] are compared one by one as we would in any other so-called dummy[7] implementation and iterate until the mismatch

[3] A string metric quantifying how dissimilar two strings are one to another.

[4] Think of it now as C representing some function and D another.

[5] A hash is a value generated using an arbitrarily chosen math function that should be unique to every variable/object being hashed.

[6] The following tokens would be tokens 9 and 10, if a detected hash were at position 1-8.

[7] In programming, the word "dummy" refers to something primitive, easily interpretable, implementable, and typical for the ones new to the subject.

or the end of the file. Otherwise, if the hash generated from B is not found in A, we can discard the hash and move on.

The computation of hashes is performance-demanding. Therefore, KRA uses the so-called Rolling hash (Rh). The principle of Rh is that the hash does not have to be computed in its entirety all the time, but the computational function is adapted to remove the first element and add a new one at the end easily. The basic idea is that if we have a sequence "abcdef," a length set to 4, and a hash function that adds the ASCII values of the letters, we write the three hashes as $abcd = 97 + 98 + 99 + 100$, $bcde = abcd - 97 + 101$ and $cdef = bcde - 98 + 102$. Thus, when counting the second and third hash, two addition operations were saved at the cost of one subtraction. The saved computational power will grow as the length of the hash grows. On the other hand, if the minimum sequence length is too small, the algorithm becomes very inefficient since we will often apply the dummy implementation and even compute the hashes as extra work.

2.3.3 Greedy String Tiling

Greedy String Tiling (GST) algorithm came to consideration as a fitting solution. See Section 3.1. GST is used, for example, by the Faculty of Information Technology of CTU in the ProgTest system as a plagiarism detector [4] or outside of the programming world to compare DNA/protein sequences [19]. For clarity, from now on, we will call the sequence of identical tokens a *Tile* instead of a match, as we have done so far, as it is used in the original paper [18]. Thus, a *Tile* refers to a sequence of tokens found in at least two files or DNA sequences. GST calculates the similarity level of two source files in two steps:

- Tokenization - created by lexical analysis in Section 2.2.
- Searching for tiles - described in this subsection

From now on, we will need to define the terms related to GST, as it is used in the original paper, to be able to explain the algorithm [18] adequately:

- Maximum match (maxMatch) - the longest possible sequence with identical tokens line in a current iteration[8].
- Tile - an identical sequence of tokens found at least at two places, which was *maxMatch* and therefore will be *Marked* for all the following iterations of GST. Remember, the tokens of one tile in two files can contain different text.
- Minimal match (minLen) - minimum length of subsequence, which we do not evaluate as noise but as valuable information in a file.
- Marked token (Marked) - a token already assigned to another tile. Each token can be a part of just one tile.

[8]Iteration means one run of a code inside a while cycle, from line 8 to 33 in Figure 2.3.

The implementation of GST is simple and can be understood from pseudocode in Figure 2.3. Initially, we set up variables to store the similarities and a record of the tokens used (lines 2-4). From that point, the program runs in iterations over and over again (lines 8-33) as long as *maxMatch* is longer than *minLen*. Otherwise, the algorithm terminates.

In each iteration, the tokens of file A are compared with the tokens of file B pairwise (lines 11-13), and if there is a match, the algorithm tries to maximize it (lines 16-19) by simply comparing tokens one by one until the mismatch or the end of the file. When the tokens are no longer equal, and the found length is greater than *maxMatch*, the current match is saved as a potential *Tile*, and the previous matches found (lines 22-27) are discarded. If multiple similarities of the same length are found, we save them all as *Tiles* (lines 29-33) and set all their tokens as *Marked* (lines 30-32) at the end of the iteration for all the following iterations.

After finding the tiles between files A and B, the level of similarity is evaluated using diceScore (dS) [1]. dS can be expressed as the ratio of the found similarities to the length of the files

$$dS = \frac{2 * (match_A + match_B)}{len_A + len_B} \tag{2.1}$$

where *match* represents the sum of the tile lengths in a given file and *len* represents the number of tokens in a given file.

It should be noted that the GST logic does not necessarily find

$$\max dS \tag{2.2}$$

that can be created between files. We will refer to *max dS* as the optimal solution, which for us means that the algorithm finds an ordering of the *Tiles* such that all possible matches longer than *minLen* are covered. *Greedy* in the name signals that the algorithm aims to produce a significant match quickly. Therefore, we will talk about the coverage found by GST as an approximation of the optimal solution that can be achieved with a fixed *minLen* [18]. However, the optimal solution to this problem in polynomial time is not yet known [4].

The approach of always finding just the longest *Tile* and marking it as *Marked* does not optimally cover the files. E.g., if *minLen* is 2, and the strings are:

A = c a a b a a d

B = b a a d c a a a a b a a

Then GST will find **a a b a a** in the first iteration and nothing in the second iteration. However, the optimal solution would be **c a a** and **b a a d**, which covers the whole A. To cover the entire A with greedy implementation, it is necessary to have *minLen* set to 1, so not only **a a b a a** would be found, but also **c** and **d**. Unfortunately, the purpose of setting the higher *minLen* was noise reduction.

2.3.4 Known GST optimizations

The basic implementation introduced in Section 2.3.3 must be optimized as it takes a long time to run in rudimentary form. Already the authors of the 1993 paper [18] came up with several improvements to make the GST run more efficiently. The worst-case complexity[9]

[9] A term used in programming jargon for the worst possible run of a program.

```
def GST(A: [tokens], B: [tokens], minLen: int, len_A: int, len_B: int):
    tiles = []  # list for found matching sequences
    A_Marked = [False] * len_A  # True if token already belongs to match
    B_Marked = [False] * len_B  # True if token already belongs to match
    maxMatch = minLen + 1  # the longest match of the previous iteration

    while maxMatch > minLen:  # it is possible to find a "valuable" match
        actual_match = []  # list for matches found in the current iteration
        max_Match = minLen  # always seeks for the longest match of iteration

        for i in range(len_A):  # iterate over tokens of file A
            if not A_Marked[i]:  # current token does not belong to any match
                for j in range(len_B):  # iterate over tokens of file B
                    if not B_Marked[j]:  # current token does not belong to any
                    ↪  match
                        match_len = 0  # length of match of tokens A[i] and B[j]
                        while i + match_len < len_A and j + match_len < len_B:  #
                        ↪  check the file overflow
                            if A[i + match_len] != B[j + match_len] or A_Marked[i +
                            ↪  match_len] or B_Marked[
                                j + match_len]:  # tokens do not match, or at least
                                ↪  one of tokens already belongs to another match
                                break # we do not iterate anymore
                            else:
                                match_len += 1 # prolong the match
                        if match_len > maxMatch:  # the length of match is not
                        ↪  sufficient
                            actual_match = [(match_len, i, j)]  # create a new list
                            ↪  with new match
                        elif match_len == maxMatch:
                            actual_match.append((match_len, i, j))
                        else:  # shorter than the longest match of iteration
                            pass  # match is being ignored, due to its length

        for len, position_A, position_B in actual_match:  # iterate over all
        ↪  the matches found in an iteration
            for i in range(len):  # set all the tokens of the match as marked
                A_Marked[position_A + i] = True
                B_Marked[position_B + i] = True
            tiles.append((len, position_A, position_B))  # save the match

    return tiles
```

Figure 2.3: Source code of the GST algorithm.

is currently O(n^3), which is not formally reduced[10] by the following changes, but practical experience of previous implementations resulted in a significant reduction.

- A for loop, whether inner or outer, is terminated when the current position of the token plus the longest *maxMatch* found so far exceeds the length of the file (implied by the greedy implementation).
- Create a list containing pairs of *Tile* beginnings and their lengths sorted based on their beginnings. If the position of the nearest *Marked* point from the current token is closer than the *maxMatch* of a given iteration, we can skip the unmarked points of that file, along with the following marked ones.
- If we find matching tokens between files, then instead of iterating from A_{a+1} and B_{b+1} to the nearest mismatch (lines 16-19), we repeat from $A_{a+maxMatch-1}$ and $B_{b+maxMatch-1}$ back to A_{a+1} and B_{b+1}. Reverse checking prevents finding sub-similarities of already found match. Because once the algorithm detects a sequence of identical tokens at position 5 to 20 in set A of length 15, it will subsequently spot a similarity from 6 to 20 of length 14, then 13, until it reaches *minLen*. Finding such sub-matches in a loop has O(n^2) complexity, where n is the range of the first match found.

2.3.5 Running-Karp-Rabin Greedy String Tiling

Experimentally, after the implementation of GST from Section 2.3.4, it is possible to get complexity O(n^2).[4]. It is already acceptable but still time-consuming in the case of long sequences of tokens. Therefore, the GST algorithm is associated with KRA (Section 2.3.2), and in this case, the observed complexity of the algorithm on various datasets is almost linear.[18]

2.4 UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction

Uniform Manifold Approximation and Projection for Dimension Reduction (UMAP) is a dimension reduction technique for creating a low-dimensional graph of initially high-dimensional data. The constructed graph aims to preserve the high-dimensional relationships even in low dimensions, typically 2D or 3D, so the data are easy to visualize [16].

To maintain these relationships, UMAP constructs a weighted graph where the edges represent the Similarity score (SC) measured by Equation 2.3 pairwise. Typically the connectedness is defined by the distance of vertices[11] within some predefined radius. The choice of the radius has a significant impact on the final outcome. Small radius results in isolated clusters, while a large one tends to connect everything. In some cases, specific points will not have any other vertices within their diameter, which can result in a loss of connectivity in low-dimension representation. UMAP overcomes this issue by applying adaptive radius

[10] There is no mathematical formula guaranteeing the upper limit of the run.
[11] One point in a graph.

based on *n'th* most remote vertice. SCs of isolated points are negligible even for n-1 closest neighbors. However, it preserves at least the initial local high-dimensional structure.

$$SC_{XY} = e^{-\frac{d(X,Y)-dtn}{\sigma}} \text{ [8]} \tag{2.3}$$

Let us denote SC_{XY} as the similarity score of Y relative to X, which can also be interpreted as the probability that there exists a directed edge from X to Y [8]. In order to calculate the SC_{XY}, certain parameters must be defined. One such parameter, denoted as n, is a hyperparameter[12] predefined by the user that represents the number of nearest neighbors desired for each point. The SC values are computed between every pair of vertices, with each vertex calculating its value with respect to every other vertex. The *dtn* parameter refers to the distance from a vertex X to its nearest neighbor, excluding itself. Notably, when computing the SC, one of the n points considered is the point itself, meaning that n represents the vertex itself and *n-1* remaining vertices. Let K_X be the set of n nearest vertices of X, including itself. Additionally, the value of σ varies for each vertex to satisfy the Equation 2.4.

$$\sum_{k \in K_X} SC_{Xk} = \log_2(n) \text{ [8]} \tag{2.4}$$

Later on, the SC_X's are all normalized. Normalization helps us to represent connectedness as a probability that two points were nearby in a higher dimension.

The neighborhood of each point is usually unique. Therefore, the SC_{AB} can vary from SC_{BA}. The UMAP solves it for every vertex pair by using the formula

$$SSC_{AB} = (SC_{AB} + SC_{BA}) - SC_{AB} * SC_{BA} = SSC_{BA}.$$

SSC represents an undirected edge between A and B, so $SSC_{AB} = SSC_{BA}$. Because of that, the global graph keeps the connectedness of A and B even when just one of them contains the other in its set of points, which is typical for isolated points, and therefore maintains the global high-dimensional structure better. Alternatively, the entire global SSC for all the vertices can be represented as

$$B = A + A^T - A \circ A^T \text{ [8]}$$

where A is a weighted adjacency matrix[13] and $\circ$ is the Hadamard (element-wise) product.

The low-dimensional graph is initialized by spectral embedding, which starts with Laplacian matrix L computed as:

$$L = A^T A = D - B \tag{2.5}$$

[12] A parameter that cannot be estimated from the data directly. (can be evaluated by cross-validation)

[13] A matrix with dimension n x n informs which vertexes are connected(filled as one otherwise zero), having zeros at the diagonal.

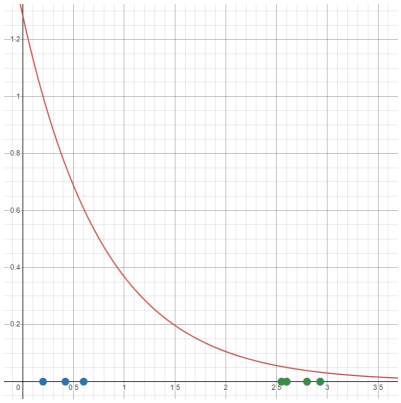


Figure 2.4: Graph representing the relationship between distances of points on the x-axis and SC on the y-axis.

where A is an incidence matrix[14], D is the degree matrix[15] and B is an adjacency matrix [17]. Once the eigenvectors have been computed, they are ordered in ascending order based on their corresponding eigenvalues. From these eigenvectors, we leave out the first one, corresponding to the eigenvalue 0, and use the desired number of eigenvectors to be selected based on the desired low dimension. This ordering ensures that the chosen eigenvectors capture the most significant variability in the data. Selected eigenvectors are then used to project high dimensional graph to the lower dimension.

A low-dimensional constellation of points generated by projection is not optimal. UMAP shifts the points applying the high-dimensional SSCs in the current dimension. The algorithm starts with randomly choosing a point A to be moved. Secondly, it picks point B from n-1 neighbors by using SSC_A as probabilities for selection. When B is selected, UMAP makes a 50/50 decision, whether to move A towards B or vice versa (in our case, let B be the vertex, which will shift). The third step is to randomly choose another point C, not from n-1 neighbors of B (chances of all these points are now equal). The goal is to get B closer to A's location but simultaneously further from C.

It is done by computing low-dimensional Similarity score (lSC) using a fixed symmetrical curve based on a t-distribution.

$$lSC = \frac{1}{1 + \alpha d^{2\beta}}$$

The parameter d represents the distance in low dimension, α and β reshape the distribution.

In our case, the lSC_A is represented by the green distribution and lSC_C by the red one.[16] B is depicted as a purple circle. lSC_{AB} and lSC_{CB} are inserted into a cost function shown in Equation 2.6 and evaluated. A decrease in cost means B was moved closer to

[14] A matrix with dimensions m x n represents a graph with m edges and n nodes. Each row in the matrix corresponds to an edge and is encoded using the values -1 and 1 to indicate the start and end vertices, respectively. The rest of the row is filled with zeros.

[15] A matrix with dimension n x n, with the number of edges connected to the vertex at the diagonal and zeros elsewhere.

[16] "A/C" will be the middle of "green/red t-distribution."

Figure 2.5: Vizualization of lSC, when α and β are set to 1.

the current optimal subposition[17]. The location of B adjusts a few times using Stochastic gradient descent until B reaches optimal position relative to A and C.

$$Cost = \log \frac{1}{lSC_{AB}} + \log \frac{1}{lSC_{CB}} \text{ [16]} \tag{2.6}$$

After these steps, UMAP selects another three points and repeats the algorithm.

2.5 HDBSCAN: Hierarchical Density-Based Spatial Clustering of Applications with Noise

Hierarchical Density-Based Spatial Clustering of Applications with Noise (HDBSCAN) combines density-based and hierarchical clustering. To diminish susceptibility to outliers, HDBSCAN starts by transforming the space and changing the metrics to compute the gap between two points. The new metric will be called mutual reachability distance, denoted as in [11]:

$$d_{mreach-k}(A, B) = \max\{core_k(A), core_k(B), d(A, B)\} \tag{2.7}$$

where $core_k(X)$ represents the radius needed to obtain k nearest points in it and $d(A,B)$ represents regular Euclidean distance. Under this metric, the dense points remain the same distance from one another. However, the sparser ones are being pushed away from other vertices, at least to their core distance away from the rest [3].[18] See Figure 2.6.

In Figure 2.6, the $d_{mreach-k}$ between blue and green will be the core distance of green, but the green-red $d_{mreach-k}$ will be their Euclidean distance.

After computing the $d_{mreach-k}$ of all pairs of points, the next step is to construct a weighted graph connecting all points, where vertices represent data points and edges the $d_{mreach-k}$. Then throw away all the edges having a weight above a certain threshold, and by iterative lowering of threshold, more and more vertices will disconnect from the graph. It forms clusters. However such an approach is computationally demanding since there are n^2 edges. Fortunately, a minimum spanning tree from graph theory offers an effective solution. Multiple algorithms can build minimal spanning trees, e.g., Dual-Tree Boruvka or Prim [3]. The spanning tree construction process is iterative and involves adding edges to the tree one at a time. The algorithm identifies the shortest edge connecting an existing tree with a

[17] If another point were selected, the direction of the slide would change, as well as a final destination.
[18] It implies that the higher the k, the more points will be clustered as sparse outliers

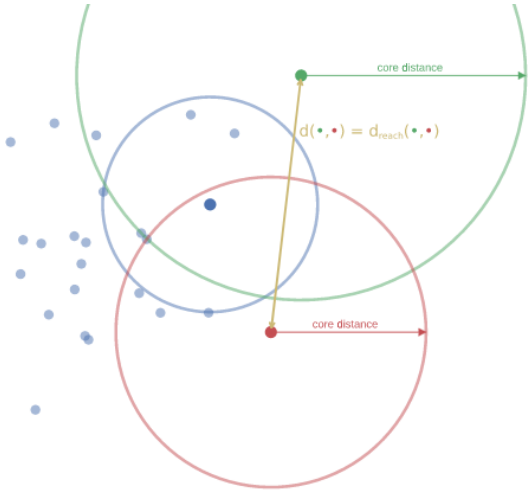


Figure 2.6: Computation of $d_{mreach-k}$ [14].

vertex not yet in the tree at each iteration. This edge is added to the tree, and the process is repeated until all vertices are included. It means that each vertex will be connected to exactly one other, so every edge removal will result in creating a new distinct group of vertices. See Figure 2.7.

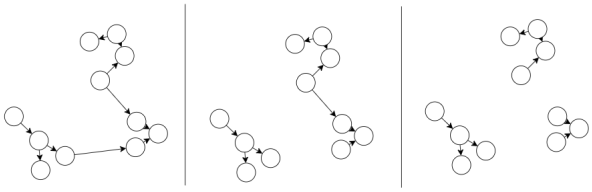


Figure 2.7: Disconnection of a minimal spanning tree.

Given the minimal spanning tree, the next step is to convert that into the hierarchy of connected components [3]. Standard Single Linkage Clustering[19] can be applied to obtain a dendogram [11]. See Figure 2.8.

Now, the clusters with a persistent lifetime will be chosen as final clusters.[20] To measure persistence, the inverse of the distance will be handy.

$$\lambda = \frac{1}{distance} \text{ [14]}$$

Then for every cluster λ_{birth} and λ_{death} is defined. λ_{birth} is the lambda (inverse of threshold) when the cluster became its own and λ_{death} when the cluster split (if ever). For each point, the value λ_p is kept as information when the point "fell off the cluster" [3]. These splits

[19] Also called Aglomerative clustering.

[20] A cluster with a persistent lifetime represents a group of points that will remain connected even after the elimination of isolated points.

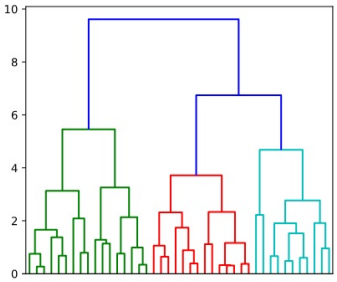


Figure 2.8: Dendogram representing clustering cut at height 6 [6].

create a tree-like structure, see Figure 2.9, where the number in the square represents the number of points in the cluster before the split. Finally, the stability of the cluster can be computed as

$$\sum_{p \in cluster} (\lambda_p - \lambda_{birth}) \text{ [14]},$$

where p in $p \in cluster$ represents the set of points that remained in the cluster until the end or until the next split.

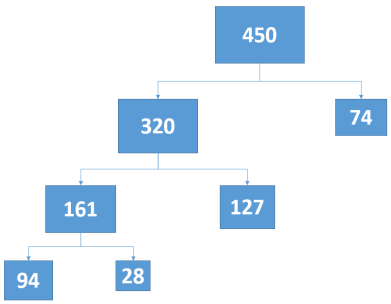


Figure 2.9: Clusters created after the elimination of the groups of points smaller than a cluster's minimum size.

In the end, it is enough to "climb up the tree" similar to Figure 2.9 from its leaves. If the sum of the persistence of the child clusters is higher than the one of the cluster, the cluster stability is set to the sum of the child stabilities [3]. Otherwise, the cluster is defined as flat. All the flat groups are then considered to be accurate and will be denoted as final clusters. The paragraph can be easily explained by Figure 2.10, where three clusters are distinguished on the left. However, on the right, only two since the peaks are probably just small subfeatures of the class.

2.6 Decision trees

A decision tree is a learning algorithm used for classification. Its advantage is transparency and interpretability, which mimics human-like thinking. See Figure 2.11. A tree

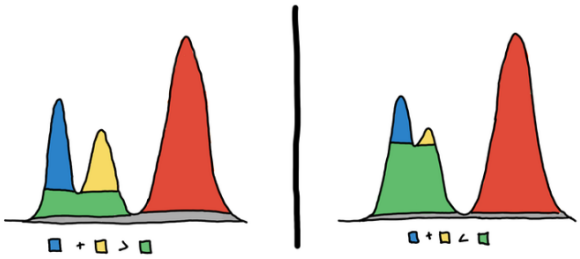


Figure 2.10: Extraction of clusters from density distribution [10].

is built by nodes representing decision attributes and leaves representing classes assigned to data belonging to them. Node order is created by recursively splitting the dataset into smaller subsets using the selection metric, which minimizes the impurity of subsets.

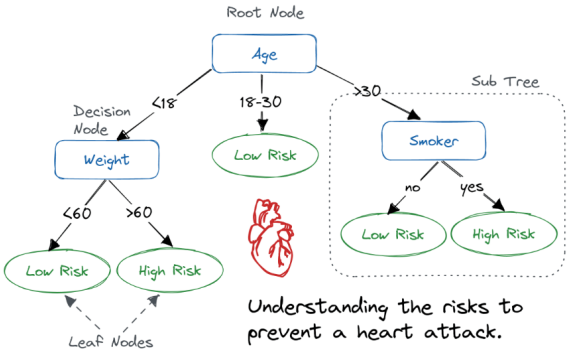


Figure 2.11: Illustration of a decision tree interpretability [2].

Therefore, a "selection metric"[21] is always chosen at the beginning. There are many of them, such as Information gain, Gini impurity index, Entropy, etc. This section will use the Gini impurity index (GII) to explain decision trees.

The starting task is to select a feature[22] that predicts the labels the best. Before GII will be formulated as a selection method, "Giny value" needs to be defined. See Equation 2.8.

$$Gini(D) = 1 - \sum_{i=1}^{n} p_i^2 \text{ [20]} \tag{2.8}$$

n represents the number of elements in node/leaf, D is a set of all elements obtained by a node/leaf containing possibly n different labels. At the same time, p_i can be interpreted as $p_i = \frac{|D^i|}{|D|}$, where D^i represents the elements of label i in D. Current knowledge is sufficient

[21] Reflects L1, L2, etc. norms when measuring distances. All norms have the same goal but with slightly different intentions.

[22] One of the attributes representing the data, e.g., age, gender, income, etc.

to define GII by Equation 2.9.

$$Gini\_index(D,k) = \sum_{i=1}^{I} \frac{|D^i|}{|D|} Gini(D^i) \text{ [20]} \tag{2.9}$$

k refers to the feature for which the GII will be computed, and in Figure 2.12, it is "Income above 50K." I is a set of labels in D, in Figure 2.12 "Yes" and "No.". $\frac{|D^i|}{|D|}$ is a normalizing factor based on subset volumes so that GII can be seen as a weighted average of *Gini values*.

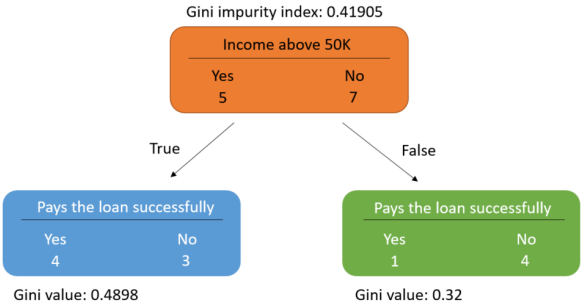


Figure 2.12: Illustration of one GII computation.

After the computation of all the GII for all the features, the feature with the lowest *Gini index* will be chosen as a decision feature. Later on, the algorithm iterates until the subsets are not pure, which means that the subset contains just data labeled as "Yes" or "No" (in our case). Such an approach can lead to overfitting[23] since the bottom-most nodes could contain just one element labeled differently from the rest of the subset, but the decision tree would still create a new branch. Without the limit, the algorithm will commonly reach accuracy near 100%, which is alarming, and it can be said that the model is overfitting the training data. These sparse leaves are commonly not considered trustworthy because so few elements can be just random/unverified data without confidence in their future impact. The algorithm overcomes it by setting up the minimum elements of a node or by pruning[24].

[23] Model predicts the data without generalization.
[24] Running cross-validations to obtain the best generality.

Chapter 3

Proposed solution

The ongoing chapter discusses the process of choosing the methods used in this bachelor thesis. Important to mention, no public solution to similar works was found, and therefore there was no option to compare or get inspired by other papers or theses.

3.1 Detection of common technical structures

Before the business logic in SQL scripts can be uncovered, common code structures must be defined. Unfortunately, a standalone file is too general to detect business logic since it can contain multiple methods programmed with numerous purposes. Therefore files needed to be split into smaller pieces, ideally methods or their subparts executing one particular action, such as building constraints in the WHERE-clause or IF-statement. Let these parts be called *snippets*.

Based on the assumption that these *snippets* appear in every method multiple times, it would be worth being able to enumerate these *snippets* and focus on the most common ones. It gives rise to two questions. How to break files into pieces, and how to compare those pieces? The university plagiarism checks of programming assignments inspired this thesis to apply a similar approach toward the bank's SQL scripts and see them as potential plagiarism.

The Plagiarism detection (PD) algorithms are well documented, and so [4] was used to understand how these algorithms work. It solved both questions raised above. PD starts with lexical analysis. It overcomes the issue of comparing the *snippets* by parsing the code into tokens, which can be compared based on their use in the source codes and not the text they are represented with. Then algorithm for matching sequences of tokens runs, and the token streams found can be considered as *snippets*.

Lexical analysis was a straightforward task, but the detection of similarities was not. [4] goes straight to the Running-Karp-Rabin Greedy String Tiling (RKRGST) implementation, but it is not the only possible solution out there.

Three algorithms met set requirements and were considered while planning the implementation. RKRGST, which was used in this research as in [4], and two lesser-known algorithms called Knuth Morris Pratt (KMP) and Boyer-Moore (BM). The first requirement was to detect transpositions of code blocks since, regardless of the order of the implemented functions,

just execution order matters. It means if we have two files, the first one contains methods in order "AB" and the second in order "BA," and we execute the call "BAB" upon both files, the results will be identical. The second constraint was the ability to detect multiple repeating patterns within one file. It slightly covers the requirement of duplicities detection. However, it is mentioned separately because, from the previous explanation, it could be understood as when the sequence is found, and if it does not appear in the second file in the form of one consecutive stream (as in file one), check whether you can break it into pieces and find them in it. In reality, it means that if the first file looks like "ABCDEF" and the second one like "BAKLDF," the *tiles* will be "B," "A," "D," and "F."

It was decided not to use BM and KMP since they are very memory intensive, and unfortunately, memory is a decisive requirement of work in provided conditions. There were 13 370 source files handed over by Česka sporitelna, some of which are a few thousand lines long, and every file needs to be compared with all the other files.

3.2 Detection of common logical structures

At the moment of having snippets, it was already possible to uncover tasks hidden in them. Again, two options were considered. The first one was embedding, a common approach of language models when building the net of dependencies of words and their combinations. The second one was to reduce the dimension of the original data and then cluster it in the lower dimension.

Embedding looks like a favorable option since it is widely used in many known and used applications. Unfortunately, it does not fit this case of work. Nothing is known about the provided data. The number of business logics, size of the business logics, anything. Therefore, the probabilistic interpretation given by embedding would be hard to understand and explain. When is the probability high enough? Isn't it high enough because the logic is comprehensive and covers multiple conditions, and thus the probability is distributed into more of them? Answers to these questions are challenging and left to be solved for the other researchers interested in embedding.

Dimension reduction also offers various approaches. In the beginning, linear methods for dimension reduction such as Principal Component Analysis (PCA) were eliminated due to a lack of knowledge about the data. In non-linear implementations, t-distributed Stochastic Neighbor Embedding (t-SNE) is considered a state-of-the-art solution, but often mentioned and cited is also Uniform Manifold Approximation and Projection for Dimension Reduction (UMAP).

How UMAP works was explained in Section 2.4. t-SNE reduces the dimension on the same principle, but two main factors influenced the decision to stick to UMAP. The main one is that UMAP scales much better on more extensive data than t-SNE. It is due to the way it shifts the points in space. UMAP constantly shifts just one point in one iteration, while t-SNE moves every point in every iteration a little bit. It makes it less accurate and more computationally demanding. The second reason was that the projection into a lower dimension is deterministic for UMAP using spectral embedding. For t-SNE, the projection is based on probabilities of point connectedness so that the results can differ slightly with every iteration. It does not mean that UMAP always offers the same result since it uses

Stochastic Gradient Descent for shifting the points in a lower dimension, but randomness influences the results later than in t-SNE.

Generally speaking, clustering attempts to group data in a way that meets human intuition [11]. As said in the first paragraph of the section, dimension reduction results will be clustered. There is still no desired number of expected classes in the data. Therefore methods highly dependent on this parameter should be excluded from the list of possible clustering methods. Otherwise, extensive cross-validation will always be necessary while clustering the data. There can be unique points, called outliers, that do not belong to any business logic group, or their logic is not commonly executed in the provided source codes. It eliminates all partitioning-based clusterings, for which it is typical that every point in space needs to be assigned to a cluster.[1] For this thesis, it is required to have accurate clusters so that the business logic can be clearly defined.

The requirements set above leave two commonly used options. Density-Based Spatial Clustering of Applications with Noise (DBSCAN) which is the state-of-the-art solution because of its robustness to various shape clustering. Unfortunately, the number of nearest neighbors is hard to predefine. Hierarchical clustering (HC), on the other hand, returns a set of nested relationships about how the clusters were created, represented by a dendrogram, so the HC is able to define how many clusters the data have. The combination of these two is HDBSCAN which is explained in Section 2.5 and was also used here.

At the point of having labels, there is no more straightforward machine learning algorithm to interpret than decision trees. And since the interpretability for the banks was a primary constraint, it was also the method used at the end of this thesis. The nodes near the root will determine a general BI group, while the leaves will specify one explicit task. Therefore, every branch will be considered one logic, and by descending the tree, the BI will be more and more defined. Based on the cardinality of nodes would be later possible to define an approximate number of main BI groups in the provided files.

[1] Typical representant of the partitioning group is the K-means algorithm.

Chapter 4

Implementation

This section discusses implementations of the previously mentioned methods, which form the basis for this work. For the purposes of this thesis, the business logic is stored in SQL scripts, over which we perform a lexical analysis (Section 4.1), then the algorithm to find similarities between the files is shown (Section 4.2). At that point, the technical structures are found. For example, it means a structure of a WHERE-statement with two conditions $A = 5$ *and* $B = \ 'cust'$ was detected, but it is not known what all the combinations of conditions are in the WHERE-statement. Two following algorithms will detect the common patterns in texts of these statements or conditionts. The dimension reduction technique (Section 4.3) will follow, over which the sorting (Section 4.4) will be performed. In the end, decision trees (Section 4.5) are applied to the previous results, where typical combinations of statements for executing business logic should be visible.

4.1 Tokenization of source codes

Creating a functional lexical analyzer for the SQL programming language would be beyond the scope of this bachelor thesis, so we decided to use the open-source library *sqlparse* [15], which is capable of analyzing most of the source code programmed in the SQL language. There were detected flaws about which the main contributors were informed. E.g., there are no functions provided that return the detected token type directly, just complex strings, of which just a part in the middle was talking about the class. It was solved by creating a parser on their parsed outcomes returning a wanted token type. Unluckily, the reported flaws went unnoticed, and no reimplementation in commits[1] was seen.

Before the execution of the parser, the code was freed from comments, whitespaces, or other parts that do not affect the execution of the program but could cause discontinuities in the evaluation of the similarities of two or more programming codes.

The introduced library is able to parse a program into tokens such as WHERE, IF, JOIN, COMPARISON, or many others. However, the library only classifies the so-called "first layer" of tokens. It can be understood that the WHERE-clause is detected, but we

[1] A commit is a record of changes made to the implementation of any application in a programming society.

know nothing about what it is made of. Typical for the SQL language are nested SELECTs[2], which can be seen as standalone functions, and thus can carry more valuable information than the standalone WHERE-clause in the code.


```
{
  "text": "from",
  "type": "Keyword",
  "subtype": "from",
  "int_to_compare": 11,
  "substatements": [],
  "has_substatements": false
},
{
  "text": "VDS_OWNER.OKO_HC_SALES_PLANS PL",
  "type": "Identifier",
  "subtype": null,
  "int_to_compare": 2,
  "substatements": [],
  "has_substatements": false
},
{
  "text": "where PL.OKOHCSP_ORG_LEVEL_NUM = 0 and pl.OKOLT_SOURCE_ID = 'PROD' ",
  "type": "Where",
  "subtype": null,
  "int_to_compare": 13,
  "substatements": [
    {
      "text": "PL.OKOHCSP_ORG_LEVEL_NUM = 0",
      "type": "Comparison",
      "subtype": null,
      "int_to_compare": 14,
      "substatements": [
        {
          "text": "PL.OKOHCSP_ORG_LEVEL_NUM",
          "type": "Identifier",
          "subtype": null,
          "int_to_compare": 2,
          "substatements": [],
          "has_substatements": false
        },
        {
          "text": "=",
          "type": "Comparison",
          "subtype": null,
          "int_to_compare": 14,
          "substatements": [],
          "has_substatements": false
        },
        {
          "text": "0",
          "type": "Integer",
          "subtype": null,
          "int_to_compare": 15,
          "substatements": [],
          "has_substatements": false
        }
      ],
      "has_substatements": true
    },
```


Figure 4.1: A part of script represented by tokens in JSON format.

The "First layer" problem was solved by recursively calling functions from the *sqlparse* library, which gave us a hierarchy of code in a tree structure. See Figure 4.1. For example, from WHERE-clause "where a = 5" was the keyword "where" removed. Then the parsing function was called on the rest, which forced it to identify COMPARISON. Then everything before "=" was taken apart, and on both parts (before "=" and after including "=") was the parsing algorithm called again. There were just a few types, such as WHERE, COMPARISON, PARENTHASES, or a few other newly created types for the need of this bachelor thesis, which were parsed further.

Then the data was stored in JSON form. A single token stores information about its type,

[2] "Select" is a statement that searches for a set of results(table rows in SQL) based on predefined constraints.

subtype[3], and the text that defines the token. If the token type is subdividable, everything about its descendants is also stored in it. An example can be seen in Figure 4.1.

4.2 Search for similarities

The ability to detect similar sequences of tokens will be crucial for this thesis since multiple files were provided. In reality, one file does not mean one business logic. Therefore smaller chunks of scripts, which appear through various files, no matter how long they are, would reflect the single business logic much better.

After the elimination of KMP and BP, the implementation of RKRGST (Section 2.3.5) was initiated. Multiple improvements were needed to compute *tiles* in a reasonable time. The first minor technical improvement was to assign to every type + subtype from Figure 4.1 a unique integer ("int_to_compare"), so comparing just two integers instead of possibly four strings was necessary.[4]

An algorithmic improvement was to apply dynamic programming principles and not compute any of the once-computed values another time. It can be understood as when pairwise comparisons are made, file A will run RKRGST on file B, but later on, file B will also call the script on A, and the results will be identical. Thus, it is necessary to compute just the "upper triangle" of a table, which holds all the results of RKRGST. See Figure 4.2. The first symmetrical table represents an unoptimized version of the computation. The

1.)	A	B	C	D	E
A	0.6	0.23	0.41	0.02	0.11
B	0.23	0.12	0.33	0.05	0.16
C	0.41	0.33	0.23	0.21	0.18
D	0.02	0.05	0.21	0.14	0.72
E	0.11	0.16	0.18	0.72	.028

2.)	A	B	C	D	E
A		0.23	0.41	0.02	0.11
B			0.33	0.05	0.16
C				0.21	0.18
D					0.72
E					

3.)	A	B	C	D	E
A	0.6	0.23	0.41	0.02	0.11
B		0.12	0.33	0.05	0.16
C			0.23	0.21	0.18
D				0.14	0.72
E					.028

Figure 4.2: The tables represent dice scores computed by Equation 2.3.3 upon *tiles* generated by RKRGST.

second table represents an optimized version of the calculation fitting the regular use of RKRGST. One file is compared to all the other files, and it makes no sense to compute dS on itself because it would be automatically 1, so the diagonal is empty. It does not apply to the case of this work, where even one file can contain duplicities in it. To overcome an evident 100% match, comparing tokens at identical positions while comparing the file to itself was denied and forced to spot duplicities at different locations if there were any. This work case is represented by table three with a filled diagonal. From the previous sentences, it is obvious that the computation of dS does not make sense, and RKRGST provides just *tiles* as business logic snippets for the following methods. Appearances of each *tile* will be summed up. Those with numerous appearances and sufficient length will be considered significant and analyzed first.

[3]It can be understood that the WHERE-clause is detected. The type keyword covers words such as "as," "from," or many others which will be assigned to JSON as its subtype.

[4]Not every type has a subtype, but after matching main types, it was required to make a boolean call and ask whether the token "has substatements." If the token has substatements, the subtypes need to be matched, and even then, the tokens are recognized as equal.

Despite the incredible efficiency of the improvements mentioned here and in [18], computation limitations were still present. It led to a file rearrangement, which was finally sufficient optimization. The files were lined up in ascending order, and then the RKRGST was called. Such alignment saved multiple iterations over hashes. Imagine an extreme case when we have 3K files containing just one hash and one file containing 50K hashes. If the long file were at the beginning, based on implementation from Section 2.3, it would iterate over 3K files, and with every iteration, it loops over its 50K hashes. It results in 150 million actions executed to compare all short files with large one.[5] Otherwise, in the case when the extensive file was at the end, we will execute just 3K actions. Consider that every action will compute at least one hash, and there are 13 370 files built in a variety of 10 to 50 000 tokens, resulting in billions of iterations.

In the end, it is relevant to mention that RKRGST is fully parallelizable. It means the current implementation is independent of previously computed results. So, if there were sufficient threads and memory, every file could be compared with all the other files in its own thread. Even more, these threads could be parallelized. Consider a scenario where three files are denoted as ABC and an unoptimized version of RKRGST. In order to compute the dS values, three threads (T_1, T_2, T_3) can be used. T_1 compares A with B and C, T_2 compares with A and C, and T_3 does it for C with A and B. It is possible to use six threads and execute each comparison in a dedicated thread, and the algorithm will still not need to wait for anything.

4.3 Dimension reducion by UMAP

Dimension reduction is the transformation of initially high-dimensional data into a lower dimension so that the low-dimensional representation retains the properties of the original data. When having complex data, there are two options. The first is to insert them into a neural network and wait for the results, which cannot be easily interpreted. Alternatively, reduce dimension since, typically, the majority of dimensionality is redundant. Pruning out the data reduces the noise and increases interpretability for analysis.

For the computation of UMAP, the python *umap-learn* [12] library was used. One-hot encoded matrix was the only data needed by *umap-learn*. Then the UMAP returned points in the desired dimension. For example, every row of the matrix, initially defined by five features, was returned by UMAP with just three values for every row, reflecting the compressed high-dimensional order.

One-hot encoding is a data representation in a 2D matrix, where the first dimension equals to the number of used elements[6] and the second one reflects the number of used features[7]. Then every matrix field is either filled with "one" or "zero." Based on whether the feature is present whether not, respectively. See Figure 4.3. It can be seen that file A contains three features out of five, and file B just two, but exactly those which file A does not have. From the matrix in Figure 4.3, a person without further knowledge of files

[5] Small files will also be matched, but the number is the same for both cases. Optimized implementation of RKRGST results in 4.5 million actions.

[6] files, tiles, etc.

[7] In our case as token types: Comparisons, Identifiers, Selects, etc.

	Customer	Loan	Acc_status	Gender	Age
A	1	1	1	0	0
B	0	0	0	1	1
C	0	1	1	0	1
D	0	0	0	1	1
E	1	1	1	0	0

Figure 4.3: 0/1 hot encoding of files and features in it.

concludes that files A and E belong together, as well as B and D. The C will be somewhere between the clusters after the run of the algorithm. The UMAP does the same but with more hyperparameters and metrics used. See Figure 4.4.

Another way to influence data is to change 0/1 encoding to 0/n, where n says how many times the feature appeared in the element. The 0/n approach caused inconsistency in large elements containing the features multiple times while the small elements just a few times. Therefore, the UMAP considered the smaller ones closer to the elements without the feature than the bigger ones. However, in 0/1 encoding, they would not be nearby in the low-dimensional space. For example, 0-2 is closer than 2-100, even though the elements with identical features incorporated even once have more in common than the others. Such an approach can be used on similarly sized elements. It can be solved by applying *log* to the values and having $0/\log(1 + n)$[8] encoding. The first appearance of a feature will influence the computation more than any other further appearance, and with growing n, the impact of repeatedly detected features decreases.

The *n_neighbors* in Figure 4.4 is self-explanatory and is explained in Section 2.4, *n_components* defines a minimal number of groupings in data. The *metric* stands for the norm, which will be used to compute distances between vertices. The *learing_rate* is a standard learning parameter influencing the impact of newly received information on already learned data.

As mentioned before, a single file contains multiple *tiles* possibly representing business logic. Thus the file was not used as an element in the matrix-hot representation. The *tiles* generated by RKRGST were used as rows, and columns used the text of tokens of the type COMPARE or IDENTIFIER since logic is primarily constrained by WHERE, IF, or ASSIGNMENT conditions. In case when there will be found 10K *tiles*, and every *tile* will have ten appearances, it means that the one-hot matrix will have 100K rows since *tile* is just a technical sequence, which can contain different text in it and the content is the goal of this thesis.

4.4 Clustering by HDBSCAN

Implementation of python library *hdbscan* [14] was used. The implementation is faster than a single run of DBSCAN and provides stable results, which should make the HDBSCAN a new state-of-the-art method. The low dimensional output of the UMAP was used as input

[8] 1 + n instead of n because otherwise, the matrix would contain minus infinity.

```
n_neighbors=10,
n_components=5,
metric='euclidean',
n_epochs=1000,
learning_rate=1.0,
init='spectral',
min_dist=0.1,
spread=1,
low_memory=False,
set_op_mix_ratio=1.0,
local_connectivity=1,
repulsion_strength=1.0,
negative_sample_rate=5,
transform_queue_size=4.0,
random_state=42,
metric_kwds=None,
angular_rp_forest=False,
target_n_neighbors=-1,
target_metric='categorical',
transform_seed=42,
verbose=False,
unique=False
```

Figure 4.4: Possible hyperparameters in library *umap-learn*.

to the HDBSCAN. The method returns labels for every matrix row, where unclustered data are labeled as "-1".

4.5 Decision features by Decision trees

The library *scikit-learn* [9] was imported and used to implement the decision trees, which will interpret BI in human-readable form.

The UMAP and the HDBSCAN are unsupervised methods that detect latent features of the data and group them. However, their interpretability was lost on the way to cluster them. It cannot be reversed back and reinterpreted. Therefore 0/1 encoding, initially inserted into the UMAP, is now input for the decision tree with labels assigned by the HDBSCAN in the previous step. Then, the iterative minimization of the *Entorpy* returns the decision features.

Two approaches were considered in the implementation. The first one was to keep just a label of one class and set all the others to zero. It almost always creates a small tree containing one or two decision features that distinguish the class from all others. See Figure 4.5. The first row in the node displays the chosen decision feature. The element goes to the right branch if it contains the feature, otherwise to the left. The second one talks about the value of the Entropy used as a feature metric. Another is about the number of elements in the current node. The next is about the label structure in the node, and the last talks about the predicted class. The more confident the node is about the label, the more glowing the node's color is.

Another approach is to choose n most significant labels, keep their original labels, and set the rest to zero. It was impossible to keep all the original labels and display them in

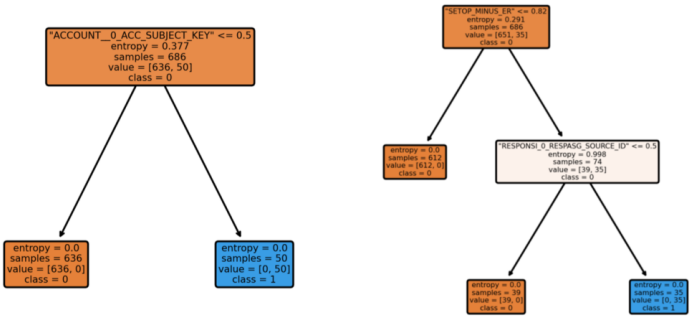


Figure 4.5: Two decision trees detect unique class features in one against all manner.

human-readable form simultaneously. Although the UMAP and the HDBSCAN have the option to set a minimum number of classes, the upper limit can go up to the number of elements provided for computation. The final tree created by the current approach creates a wider and more varied tree, often using decision features from the first approach but on a broader scale. See Figure 4.7. Therefore multi-class decision tree will be used, and every class will be displayed by a different color.

It is relevant to mention that the tree's right side is often more interesting since there are leaves with more valuable features than those on the left, where it is known what they do not contain but not known what they do. It means the purple leaf in Figure 4.7 says more than the most left leaf even though the most left one contains the highest number of elements in that level of the tree. An extreme case in data is depicted in Figure 4.6, where it was not possible to tell much since the only blue leaf represented just two elements of the class, and all the other elements were diving deeper into the overpopulated left branch.

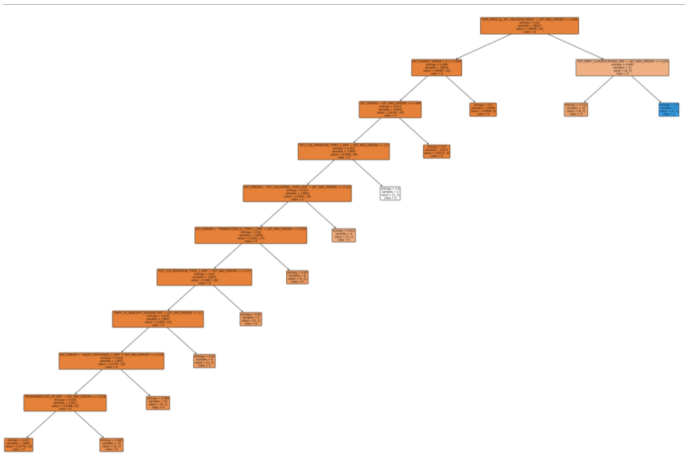


Figure 4.6: Decision tree without valuable information.

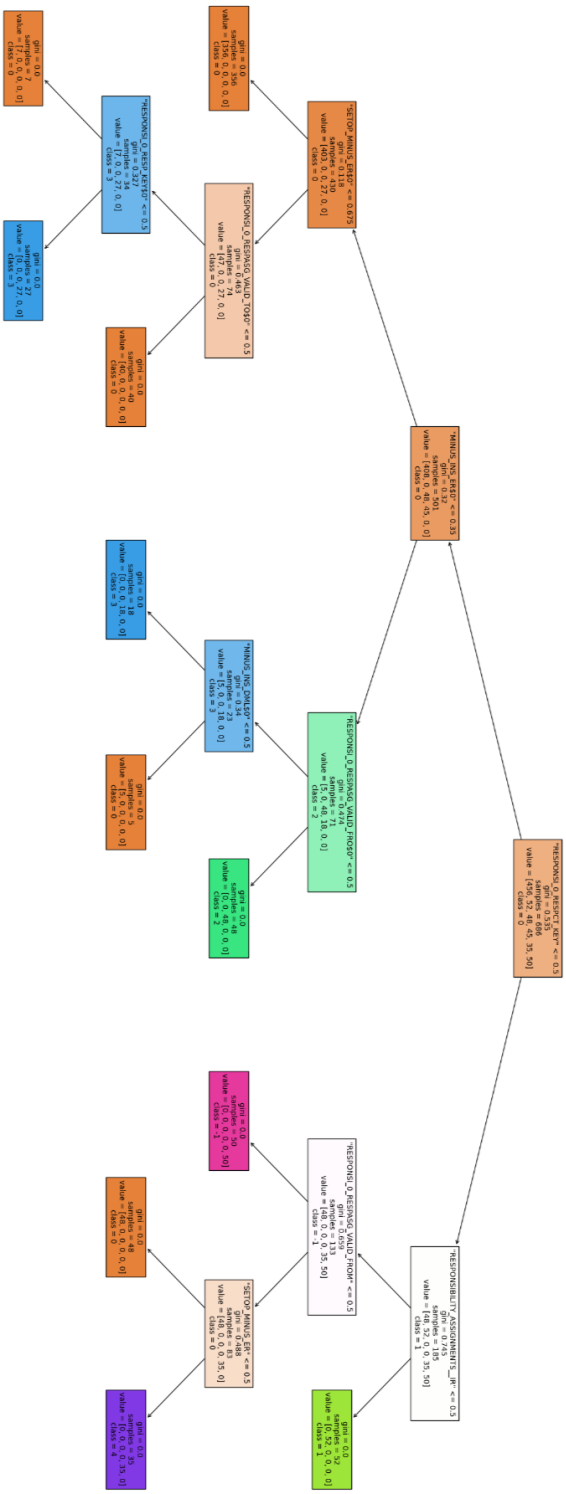


Figure 4.7: Two decision trees detect unique class features in one against all manner.

Chapter 5

Achieved results

In this thesis, the SQL scripts of the Česká sporiteľna were analyzed. Thirteen thousand three hundred seventy files were obtained. The source codes were developed over the last couple of decades, ranging from a few lines, capturing a single functionality, to thousands of complex lines describing complex data transformation operations. The ongoing chapter discusses the results achieved by the proposed solution in Chapter 3.

In the beginning, it is crucial to mention that the detected BI is a bank's private know-how. Therefore, the results cannot be displayed as a whole. Due to that reason, the chapter figures will always show just a tiny part of the tree and the logic in it.

5.1 The results of lexical analysis

The lexical analysis generated 6 782 908 tokens divided into 33 primary types and 393 subtypes, some of whose frequencies can be seen in Figure 5.1. The "NA" in subtypes means the *type* is no longer divisible and provides a complete token description, or its division was not considered necessary.

	type	subtype	N
1	Identifier	NA	1 609 807
2	Punctuation	NA	1 040 408
3	Comparison	NA	616 475
4	Assignment	NA	577 621
5	Keyword	then	276 680
6	Keyword	if	209 479
7	Keyword	end if	209 467
8	Function	NA	174 206
9	Keyword	and	144 923

Figure 5.1: Frequencies of the most common types together with their subtypes.

5.2 Generated matching sequences by RKRGST

The execution of RKRGST generated *tiles* of the following characteristics:

- 237 870 unique *tiles*
- The longest *tile*: 10 745 tokens
- The most frequent *tile*: 55 166 appearances

The data was plotted on a graph whose x-axis represents the logarithmic length of the *tile* and whose y-axis represents the logarithmic frequency of the *tile*. See Figure 5.2. Then the most significant[1] *tiles* were distinguished from the other. It was done using no algorithmic metric to quantify the outlier data, but a straight line was laid to pass through the longest and most numerous points. The points furthest from the line above were considered significant and ordered based on the decreasing distance from the line.

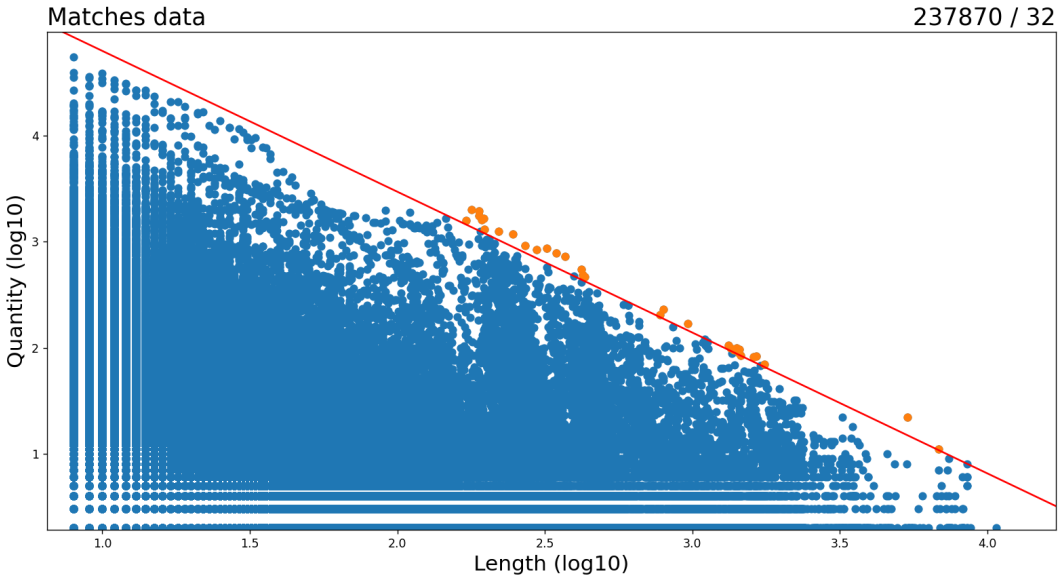


Figure 5.2: The visualization of the tiles.

Found *tiles* fulfilled the predictions and even significant points with identical token stream contained different IDENTIFIERs and COMPARISONs in it.

5.3 Execution of business logic detection

It was decided to consider only two token types, the COMPARISON and the IDENTIFIER, as possible BI holders. Even when it was said earlier that one standalone file does not

[1] Data having too high frequency with respect to its length.

represent logic, the UMAP was run with files as elements and COMPARISONs/IDENTIFIERs as features to determine whether groups of files with a similar target exist. In Figure 5.3, it is possible to see that pretty clear clusters exist. Unluckily, clusters were created upon too many COMPARISONs(some over one hundred) to call something a logic holder.

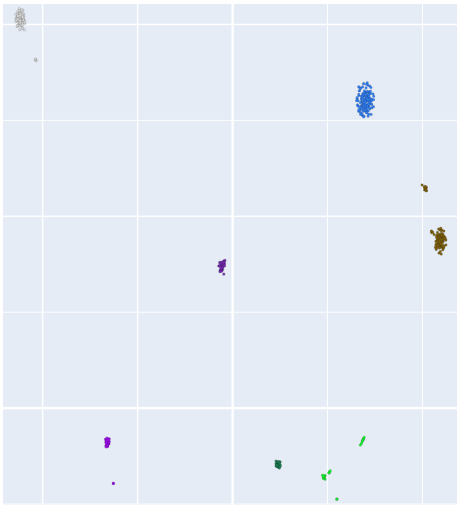


Figure 5.3: Classification of files with comparisons.

After this, another approach was applied. The goal was to know whether a relationship exists between the clustering done over the largest files with *tiles* first and comparisons right after that. So two UMAPs and clustering were executed. The first one used one-hot encoding of files as elements and comparisons as features in 3D. Later UMAP with the one-hot encoding of files and *tiles*, respectively, was initiated in 2D, and the results were combined. The 3D points from the first run were transformed into RGB[2] representation and the 2D points were colored by generated RGB colors since one point represented the same file in both UMAPs. See Figure 5.4. It showed that there does not exist correlation between technical *tiles* and logical COMPARISONs. It consolidated our assumptions that one file does not mean one BI, and from now on, the elements of one-hot encoding will always be represented by *tiles*.

As the text above suggests, the decision to use only COMPARISONSs was made. There were two main reasons. Many identifiers were defined at the beginning, but not all were used later in the file, and it just increases the volume of features for no logical reason. Also, single *Customer key* as *Identifier* does not hold BI. Logic is saved in the value in it, such as $Customer\ key = 5$ can represent the retail customers accounts, while on the other hand, $Customer\ key = 3$ can speak for the business accounts. Due to a smaller feature volume, the noise was reduced, and the calculations were sped up.

With the *tile*/COMPARISON approach, the 300 most significant *tiles* were loaded into a one-hot matrix and executed to obtain labels. Then multi-class decision tree was triggered. See Figure 5.5. Everything looked as expected, and the decision features were analyzed. The tree was built based on pure technical stuff, such as loading checks for null values or

[2]The representation of red, green and blue colors within the range from zero to two hundred fifty-five.

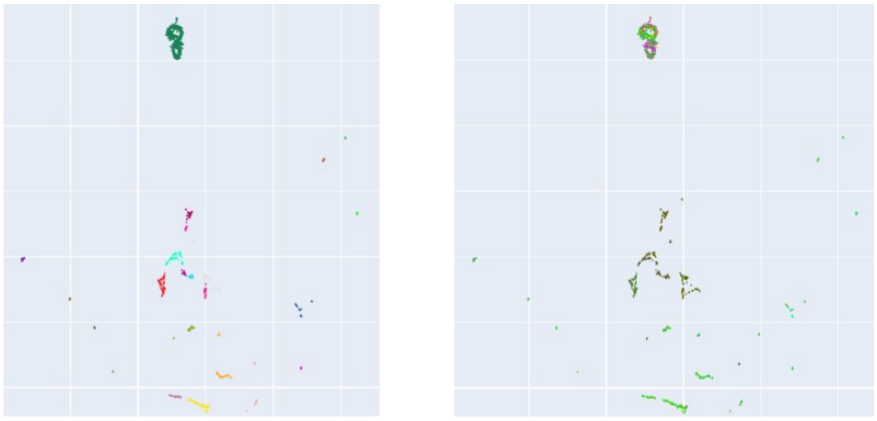


Figure 5.4: Classification before applying the 3D points into RGD and after on the right.

checks for a sufficient amount of loaded lines. These checkups are typical for almost every file, so their detection ensures the algorithm works as expected, and after a small feature engineering, obtaining valuable insight into the BI is possible.

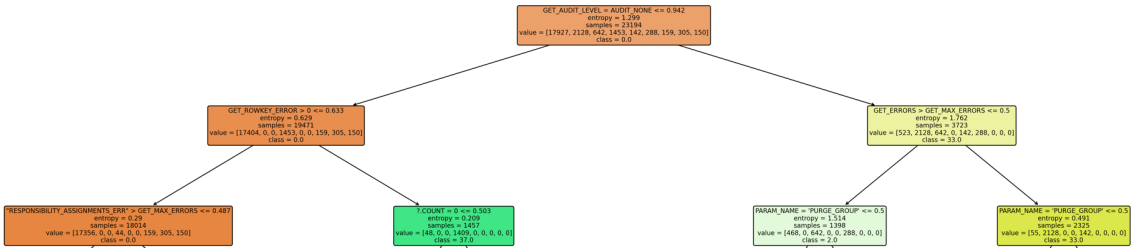


Figure 5.5: The tree's root after the tile/Comparison execution, before the technicalities were removed.

After the first run, the features containing substrings like "ERROR", "PARAM", and a few others were removed due to their high technical background. It resulted in more explicit decision features but still needed to be less technical. Therefore, one thousand of the most frequent COMPARISONs were given to a more experienced person in the bank's SQL, who provided us with "Yes/No" labels on whether it is a feature containing logic or just technicality. Another practical constraint was to limit the length of the *tiles* based on a number of tokens. The longest ones were in the majority of the times also technicalities since copy-pasting in these situations is common practice, and to have the BI of these lengths would be surprising.

These constraints caused the tree to have leaves filled with *tiles* of tokens of type COMPARISON in WHERE statements, such as $Cust.key = 5$ *and* $cust.key = ret.key$ *and* $cust.key = bb.key$ or containing few small selects. See Figure 5.6. It looks like a great result. However, two of the three COMPARISONs in the previous sentence did not give us any information because the second and third COMPARISON joined some tables without further information. This led to the last improvement of constraints on type *Comparison*. It was to

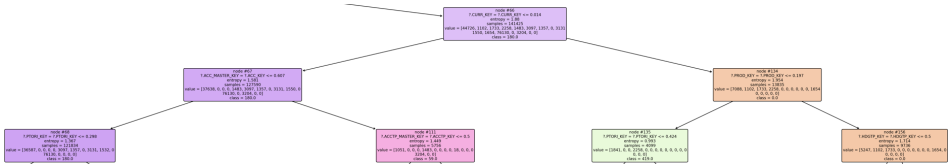


Figure 5.6: A tree branch full of join statements.

eliminate all the "joinings". At that moment, the found branches could finally be considered BI tasks. See Figure 5.7.

Later, the ranges for executed *tiles* needed to be changed because it was found that all the 200 most significant *tiles* were pure technicalities, and there was nothing to compute after removing the technical stuff. COMPARISON reduction enabled the algorithm to run over 60K *tiles* since not many contained COMPARISONs not eliminated by the constraints. Even more, reduction came when the requirement for having at least three features in every *tile* also after the elimination process. More is needed since a BI is always a combination of multiple conditions.

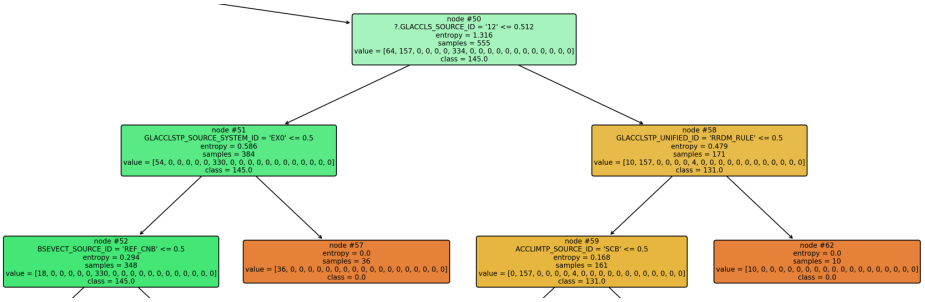


Figure 5.7: A tree branch considered as BI.

The number 60K in the previous paragraph is not accidental. At that point, the *tiles* were just sixty tokens long and appeared six times in all the files combined. Therefore it was decided not to dive deeper and have a closer look at more significant matching sequences.

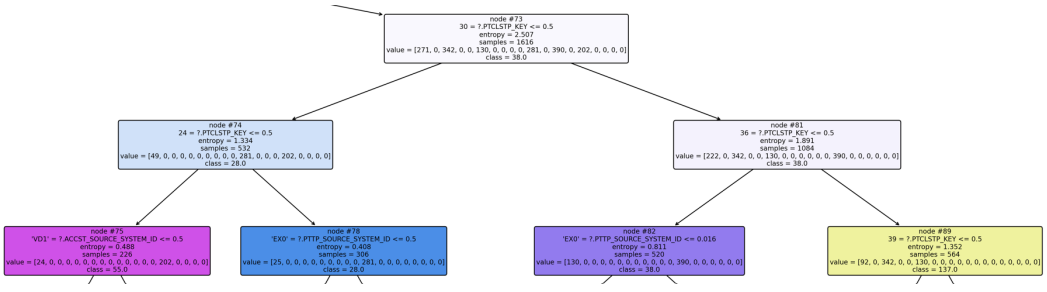


Figure 5.8: A tree branch containing new BI.

The execution of all the 60K *tiles* at once resulted in BI detection. See Figure 5.8. In reality, it had not changed from the 20K computation much, which was triggered before ($20K \sim 60K$). It is because the tree was biased towards the most prominent groups detected at the beginning since their decision features reduce the entropy the most. The smaller groups were overshadowed by the bigger ones. It led the algorithm to iterative runs on different ranges, such as 20K to 40K or 40K to 60K, which resulted in the detection of new combinations ($[20K\ to\ 40K] \nsim [40K\ to\ 60K]$).

Chapter 6

Discussion

The obtained results are promising, but there are a few things to keep in mind. The current tree detects either constraints of WHERE-clauses or small SELECTs passed one after another. Both statements are BI holders, and therefore it is something wanted. Unfortunately, their weight in computation comes from the implementation of RKRGST where in Figure 2.3 at line 24, all the potential *tiles* are saved. It causes many duplicities.

```
1 where
2     A > B and
3     B = C and
4     D = F and
5     J = Z and
6     P = H
```

Figure 6.1: A SQL WHERE-clause with multiple conditions.

In Figure 6.1, if RKRGST detects sequence from lines 2 to 4, it also sees similarity from 3 to 5 and 4 to 6. Since all of them have the same length, all three will be kept as *tiles* and saved under the same hash value. Condition $D = F$ at line four will be stored in three *tiles*. The files provided to this thesis commonly contain WHERE-clauses with 15 or more conditions and thus cause a big overheat of some sequences. If counting dS from Equation 2.1, it means instead $match_A = 5$ of conditions coverage in Figure 6.1, it would evaluate it as $match_A = 9$ and dS value for two identical SQL snippets would be higher than one. However, it is a two-edged sword. Due to the duplicities, the algorithm focuses on valuable parts of scripts and shadows all the others.

One would say it can be easily solved by always saving just the first *maxMatch* and letting the others be found in the next iteration. In the case of Figure 6.1, it would result in not having $J = Z$ and $P = H$ in any *tile* if the *minLen* would equal to three COMPARISON tokens. Or the last two rows would represent different *tile*, and the logic of these two *tiles* would never connect because, typically, these long WHERE conditions start with JOINs, which are excluded in the UMAP preparation and just later, in the end, conditions are checking concrete values or strings.

One of the other possible improvements is based on the prior[1] knowledge. The current implementation searches for *tiles* without prior knowledge, even the general one for all the SQL scripts, such as BEGIN ends with END. It does not make sense to have *tiles* searching for BI, which surpasses the END statement. The END is an obvious end of some execution and the border of the possible BI. Thus it is not necessary to continue further. It would shorten and reduce the number of *tiles* and save a lot of computational power and time. Moreover, the Česka spořitelna uses typical comments to distinguish the functions. On the other hand, as said in Section 5.3 and shown in Figure 5.3, there exists some observable relation between the file and its COMPARISONs. But it is more typical work for embeddings and research with little, but still a slightly different goal than the current one.

[1]Knowledge about the files, their structure, typical business shortcuts, etc., which is later used in evaluation to improve learning and specify the expected results.

Chapter 7

Conclusion

The research aimed to create an algorithm to cluster various business logic types. The achieved results prove the possibility of detecting business logic even in unsupervised conditions in a purely algorithmic way by creating a stream of methods with outputs serving as inputs for the following ones. Since it is not a publicly known area of research, the methodology can provide a basic approach for future researchers to investigate the logical structure of programming languages with a detailed explanation of the pros and cons of possible algorithms.

A slight limitation of this work is the inability to detect BI directly. It is necessary to have at least a few iterations with a customer when the explored BI combinations are reduced. A predefined set of substrings can help, as well as JOIN rejection, but the algorithm still cannot be certain about the results. It is typical for unsupervised learning, but the number of iterations cannot be defined in advance and can range to tens of units.

Further research should start by selecting a different method to generate *tiles*. The currently used algorithm, the RKRGST, has drawbacks, resulting in a short-term advantage. However, it is because of the type COMPARISON considered a business intelligence holder, which was a part of duplicities. With a different goal to be detected, overcoming it would be challenging.

Bibliography

[1] AJMAL, O. et al. EPlag: A two layer source code plagiarism detection system. In *Eighth International Conference on Digital Information Management (ICDIM 2013)*, s. 256–261, 2013. doi: 10.1109/ICDIM.2013.6693984.

[2] Avinash Navlani. *Decision Tree Classification in Python* [online]. 2023. [cit. 04. 04. 2023]. Dostupné z: <`https://www.datacamp.com/tutorial/decision-tree-classification-python`>.

[3] Campello, Moulavi, and Sander. *How HDBSCAN Works* [online]. 2016. [cit. 04. 01. 2023]. Dostupné z: <`https://hdbscan.readthedocs.io/en/latest/how_hdbscan_works.html`>.

[4] DVORAK, M. Detection of cribbed programs. Bachelor's Thesis, May 2022.

[5] GáLA, L. et al. *Podniková informatika: počítačové aplikace v podnikové a mezipodnikové praxi*. Praha : Grada Publishing, 3th aktualizované vydání edition, 2015. ISBN 9788024754574;8024754576;.

[6] Has QUIT–Anony-Mousse. *Interpreting Dendrogram for hierarchical clustering.* [online]. 2018. [cit. 04. 01. 2023]. Dostupné z: <`https://stackoverflow.com/questions/50894183/interpreting-dendrogram-for-hierarchical-clustering`>.

[7] HIPPISLEY, A. R. *Lexical Analysis.* 66. Linguistics Faculty Publications, 2010. Dostupné z: <`https://uknowledge.uky.edu/lin_facpub/66`>.

[8] MCINNES, L. – HEALY, J. – MELVILLE, J. UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction, 2020.

[9] PEDREGOSA, F. et al. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research.* 2011, 12, s. 2825–2830.

[10] Pepe Berba. *A gentle introduction to HDBSCAN and density-based clustering* [online]. 2020. [cit. 04. 01. 2023]. Dostupné z: <`https://pberba.github.io/stats/2020/07/08/intro-hdbscan/`>.

[11] LIBRARY, P. Hierarchical Density-Based Spatial Clustering of Applications with Noise. https://github.com/scikit-learn-contrib/hdbscan/blob/master/docs/how_hdbscan_works.rst, .

[12] LIBRARY, P. Uniform Manifold Approximation and Projection for Dimension Reduction. https://github.com/lmcinnes/umap/blob/master/doc/basic_usage.rst, .

[13] RANI, S. – SINGH, J. Enhancing Levenshtein's Edit Distance Algorithm for Evaluating Document Similarity. In SHARMA, R. – MANTRI, A. – DUA, S. (Ed.) *Computing, Analytics and Networks*, s. 72–80, Singapore, 2018. Springer Singapore. ISBN 978-981-13-0755-3.

[14] RICARDO J. G. B. CAMPELLO, D. M. . J. S. HDBSCAN: Hierarchical Density-Based Spatial Clustering of Applications with Noise. https://hdbscan.readthedocs.io/en/latest/how_hdbscan_works.html.

[15] LIBRARY, O. Parse SQL. https://github.com/andialbrecht/sqlparse.

[16] STARMER, J. *UMAP: Mathematical Details* [online]. Dostupné z: <`https://www.youtube.com/watch?v=jth4kEvJ3P8`>.

[17] STRANG, G. *Linear Algebra and Learning from Data*. Wellesley-Cambridge Press, 2019. Dostupné z: <`https://books.google.cz/books?id=L0Y_wQEACAAJ`>. ISBN 9780692196380.

[18] WISE, M. String Similarity via Greedy String Tiling and Running KarpRabin Matching. *Unpublished Basser Department of Computer Science Report*. 01 1993.

[19] WISE, M. J. Running Karp-Rabin Matching and Greedy String Tiling. 2003.

[20] YUAN, Y. – WU, L. – ZHANG, X. Gini-Impurity Index Analysis. *IEEE Transactions on Information Forensics and Security*. 2021, 16, s. 3154–3169. doi: 10.1109/TIFS.2021.3076932.

[21] ZEH, N. REGULAR LANGUAGES, FINITE AUTOMATA, AND LEXICAL ANALYSIS. Dalton University lecture, Winter 2018.

Nomenclature

BI Business intelligence

BP Boyer-Moore

DBSCAN Density-Based Spatial Clustering of Applications with Noise

dS diceScore

GII Gini impurity index

GST Greedy String Tiling

HC Hierarchical clustering

HDBSCAN Hierarchical Density-Based Spatial Clustering of Applications with Noise

KMP Knuth Morris Pratt

KRA Karp-Rabin Algorithm

LCS Longest common subsequence problem

LD Levenshtein distance

lSC low-dimensional Similarity score

PCA Principal Component Analysis

PD Plagiarism detection

Rh Rolling hash

RKRGST Running-Karp-Rabin Greedy String Tiling

SC Similarity score

t-SNE t-distributed Stochastic Neighbor Embedding

UMAP Uniform Manifold Approximation and Projection for Dimension Reduction