Master in High Performance Computing

Detection of recurring behavior in banking data

Supervisor(s):
Dr. Giangiacomo Sanna,
Ivan Girotto

Candidate:
Nesrine Yousfi

6th edition
2019–2020

Contents

List of Figures

Acknowledgment

I would like to thank the organizers of the MHPC program in both ICTP and SISSA that provided this opportunity and supported me. I acknowledge my supervisors Dr. Giangiacomo Sanna and Ivan Girotto for their supports and helps during the master project. I thank PROMETEIA company for accepting me as an intern and providing the data. I thank two anonymous reviewers for their remarks and suggestions. Also, my classmates and friends in this master's program helped me and made the time pass better despite the pandemic. Finally, I must express my gratitude to my father, sister, and fiancé for their continuous encouragement and support through all this time. This accomplishment would not be possible without all these people and organizations. Thank you.

I dedicate this work to my mother.
Rest in peace Mama.

Chapter 1

Introduction and problem statement

A recurring event is an event that occurs at regular time intervals. The detection of such events can become a difficult task, especially when it comes to covering all possible recurring scenarios.
Knowing the recurrent events and their periods allows us to predict their behavior in the future and therefore minimize the surprises and better planning our actions.
For companies like banks, insurances, corporations, etc. knowing the recurring behavior in customers' data can help understand the needs of each branch to have a better plan to support their clients (i.e. planning their loans), and/or stop them from illegal actions (i.e. money laundering). Plus, it is a way for those companies to target categories of customers for their proper offers and advertisements.
In this project, we look for recurring transactions, that is to say, a collection of transactions for an account or a card that happens regularly with almost the same amount.
Several techniques have been developed to store, read, visualize and analyze the data, with detecting such recurring behavior, especially for big data. Techniques from rule-based to machine learning that have been recently developed are explained and their advantages and lacks are mentioned in this project.
In this project, we aim to investigate different methods for data visualization, analysis, and recurring detection. In chapter 2, we will explain the main software and hardware that we use to do this project.
In chapter 3, we will discuss the steps in preparation, visualization, and analysis of two banks' datasets.
In chapter 4, we will show techniques of rule-based algorithms that are

used to detect the recurring behavior in two datasets analyzed in chapter 3.
In chapter 5, we will show how the machine learning algorithms can detect the recurring behavior in the same datasets.
Finally, in chapter 6, we present our results as a free package called bankdatainvestigation that is put on the PyPI library. This package can be used to apply all mentioned methods to detect recurring behavior in banking data.

Chapter 2

Software environment and hardware

In this chapter, we present the tools we used in this project. This software is state-of-the-art in the community of data science in data exploration, visualization, and processing big data. In the section "Software" 2.1, we talk about the used libraries and describe their main properties. In the section "Hardware" 2.2 we give details about the machines we used for our investigation.

2.1 Software

The project is carried out using Python 3.8. The main libraries that were used are pandas for data analysis, matplotlib for data visualization, scikit-learn for clustering and PySpark for scalability. Overviews for each of those libraries are available in sections 2.1.1 for pandas, 2.1.2 for Matplotlib, 2.1.5 for scikit-learn, and sections 2.1.3 and 4.3 for PySpark.

2.1.1 Pandas

Pandas is a Python software that is specialized in data analysis [**1**]. It is built on NumPy and Matplotlib libraries.
Pandas aim to be the best statistical tool that is also efficient, easy to use, and flexible. The software has been in existence since 2008 but, its development has greatly accelerated. Pandas do not live isolated and serve as a base or as a complement to other software, whether it is to manipulate geographic data, to make statistics, or to tend to replace matplotlib for certain uses.

One of the strengths of Pandas is its ability to work with several types of data in the form of tables with heterogeneous typing columns, ordered and unordered time series data, arbitrary raster data with row and column labels, and any other form of observational/statistical datasets. Even unlabeled data can be placed in a Pandas data structure.
Series and DataFrames are the basic structures used by Pandas. A "DataFrame" object allows cleaning, filtering, preprocessing, and many operations that are performed before statistical modeling. Additionally, it brings groupby functionality to perform split-apply-combine operations on datasets.
Pandas is widely used in financial applications because it is an important part of the Python statistical computation ecosystem. It is made to be the best high-level building block for doing data analysis in Python (McKinney et al., 2021).

2.1.2 Matplotlib

Matplotlib is a Python library that allows you to draw graphs, initially inspired by Matlab plotting functionalities in 2003, now is the most popular plotting library for Python [**2**].
It is well documented and, it supports various bitmap (png, jpg, gif) and vector (ps, ps, svg) file formats. Matplotlib Figure uses one of the supported user interface backends such as Qt, WxWidgets, TkInter, or MacOs. It abstracts various elements of a plot by defining a set of objects. It starts with the top-level Figure object that may contain a series of intermediate level objects and Axes – from Scatter to Line, Marker, and Canvas.
The main feature of Matplotlib is the PyPlot which allows users to write short procedural code. However, due to performance issues, it is recommended by the Matplotlib user-guide to use PyPlot only for the creation of figures and axes, and, to do the rest of the plot by their respective methods.

2.1.3 PySpark

Apache Spark

Apache Spark, created by Matei Zaharia in 2009, is a distributed framework capable of analyzing big data by processing it in parallel [**6**][**7**]. It is a framework written in Scala, which follows on from Hadoop. It offers interfaces for Python, Scala, Java, R, SQL languages.
Apache Spark is much faster than Hadoop for processing large-scale data.

It is also fast if data are on the disk. Spark has the world record for large-scale disk sorting. It can scale up to 8,000 nodes.

Spark Driver The Spark Driver is in charge of instantiating the spark session at the beginning of a Spark application, which is a connection to a Spark client that can be a cluster or computer. The Driver takes all the requested transformations and actions and creates a directed acyclic graph (DAG) of nodes. Each DAG node represents a transformational or computational step that includes schedules of running tasks. The driver coordinates the execution of stages and tasks defined in the DAG. It Keeps track of available resources to execute tasks and schedule them to run "close" to the data where possible (called data locality). Besides, it is responsible for returning the results of an application.

Spark Workers and Executors Spark Executors are the processes running on a worker node that is defined in Spark DAG tasks. They reserve CPU and memory resources on slave nodes, or Workers, in a Spark cluster. Often a spark job runs in parallel many Executors. Workers and Executors are aware only of the tasks allocated to them, whereas the Driver is responsible for understanding the complete set of tasks and the respective dependencies that comprise an application.
Java virtual machines (JVMs) host Spark Executors. They are allocated a heap memory area. The amount of memory committed to the JVM heap for an executor is set by the property spark.executor.memory or as the –executor-memory argument to the PySpark, spark-shell, or spark-submit commands. Finally, executors may store output data from tasks in memory or on disk.

Apache Spark APIs There are three sets of applications programming interface (APIs) for Spark, see [**Spark APIs**]: RDD, DataFrame, and Dataset, that have been added to Spark respectively. They have a different level of abstraction, usage, and performance. RDD is mainly used when one wants to do low-level transformation and actions on data, use the unstructured data, eliminates the optimization and performance benefits of the DataFrame.
DataFrames, higher-level abstraction. The data is organized into named columns. It is made for processing large datasets. And finally, Dataset has the highest level of abstraction among these three and it provides an optimized API. The last two APIs are built on top of RDDs. A difference between is static-typing and runtime safety. For the syntax errors, RDD's

ones will be known in the run time. On contrary, the DataFrames and Datasets will show in the compile time. However, for the analysis errors, both RDDs and DataFrames will show errors in the run time in contrary to Datasets.

PySpark

PySpark is a combination of Python and Apache Spark a Java virtual machine (JVM) based computing framework. It is a fast, user-friendly, open-source cluster-computing framework that provides streaming analytics. It has the simplicity of Python and the power of Apache Spark to work with Big Data, see [**Apache Spark**].

Figure 2.1: PySpark

Python is a general-purpose, high-level programming language. The choice of Python+spark has some advantages: Python is very easy to learn and implement, It provides a simple and comprehensive API. With Python, the readability of code, maintenance, and familiarity is far better. It provides various options for data visualization, which is difficult using Scala or Java. Python comes with a wide range of libraries like NumPy, pandas, Scikit-learn, seaborn, matplotlib, etc. It is backed up by a huge and active community.

PySpark Application PySpark enables direct control and interaction with Spark via Python. In PySpark, two separate processes run in the executor, a JVM that executes the Spark part of the code (joins, aggregations, and shuffles), and a Python process that executes the user's code. The two processes communicate via the framework Py4J that works as a bridge and exposes the JVM objects in the Python process and vice versa, see [**py4j**].

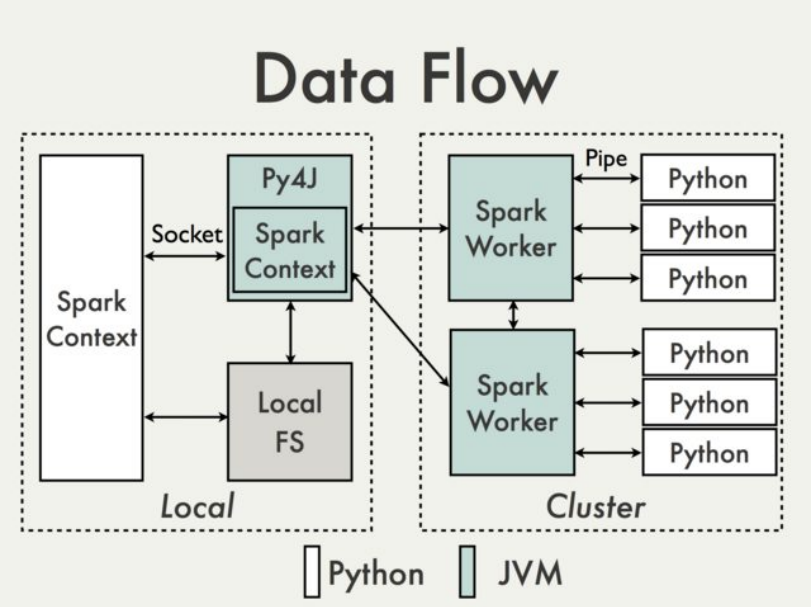

Figure 2.2: PySpark internals, [**EPFL Spark**]

2.1.4 Anaconda

Anaconda is a free and open-source distribution of Python and R programming languages. It is used to install their related packages, and it is a manager of packages and environments that allows everyone to install their Python environment (s) (or R, ruby, ...) and provides hence the following advantages:

- generate stable environments over time (therefore better reproducibility of calculations),
- the environment is exportable (simple text file) to be transmitted to a collaborator,
- autonomy from the platform's system administrators,
- this is sometimes the only solution to resolve incompatible dependency problems between users of the platform.

Custom packages can be made using the conda build command and can be shared with others by uploading them to Anaconda Cloud, Python Packages Index (PyPI), or other repositories and vice versa. In particular, packages in PyPI may be installed into a conda environment using pip

and conda will keep track of what it has installed itself and what pip has done.
It aims to simplify the management and deployment of packages. The Anaconda distribution is used by over 6 million users and includes over 250 popular data science packages suitable for Windows, Linux, and MacOS.

2.1.5 Scikit-learn

Scikit-learn is a Python library known as an integrator of a wide range of reliable machine learning algorithms. It is used for medium-scale supervised and unsupervised problems. It is a user-friendly package that offers machine learning to non-specialists using Python language. It is well-documented (over 300 pages user-guide as well as more than 60 examples), has reasonable performance (paralleled algorithm), and provides API consistency. It benefits from the BSD license that encourages academic and commercial users, see [**Scikit-learn**].
Having a rich environment, Scikit-learn provides a solid implementation for many known machine learning algorithms. In recent years, a need to do statistical data analysis by non-specialists in different fields (i.e. software and web designers in computer sciences, as well as other fields like biology, physics, etc) has been covered by Scikit-Learn.
It is a compiled code and only has two dependencies NumPy and SciPy, which facilitates its easy distribution.
Besides being written mostly in a high-level language, Scikit-learn maximizes computation efficiency. For example, it is written in a way that provides 40 percent lower memory copies than the original libsvm Python bindings. Also, they use better the pipelining capabilities of modern processors. It will be used in chapter 5 we will show how to implement Scikit-learn to detect anomalies in banking data.

2.2 Hardware

This project was done with two laptops from PROMETEIA company and a personal mac. The one of PROMETEIA company is a Dell XPS 13 (9350) which has a Windows operating system with core i7 processors 2.2 GHz up to 3.2 GHz, 8 GB of Ram, and 250 GB of SSD disk.
The personal laptop was a Macbook pro 2017 dual-core i5 2.3 GHz up to 3.4 GHz, 8 GB of ram, and 250 GB of SSD disk.

Chapter 3

Data analysis

In this chapter, we introduce two datasets that are used in our project. The first one is the Berka dataset and the second one comes from PROMETEIA company.

3.1 Datasets description

3.1.1 Berka dataset

The Berka dataset contains a collection of transactional information from a Czech bank, using Czech Koruna (CZK) currency. It is over 400 MB and deals with more than a million and a half transactions (in rows) related to banks account spread over six years, from 1993 to 1998. Berka's dataset is divided into two categories "Credit" and "Withdrawal". Among these rows, we identify 4500 distinct accounts. They contain thirteen columns, among which the following four are important for us:

- account_id: Account the transaction is issued on
- trans_date: Date of transaction, In the form: YYMMDD
- trans_amount: Amount of Transaction
- trans_type : debit/credit transaction.

In fact, the "account_id" column specifies the membership of the transaction. The "trans_amount" column contains the amounts of transactions that are important information in typifying a recurring behavior as well as the "Transaction date" column, which gives the date of transaction. Also, the "trans_type" column gives a piece of information that will determine the output of the algorithm as we will see in chapter 4.

The transactions in the Berka dataset help in taming Pandas and all the other software used in this project, like Matplotlib and Sklearn, etc (see chapter 2), and allow us to develop the techniques and algorithms for the detection of recurring behavior.
In the following table, we present some statistics on the amount of transactions column. We see that the maximum amount is 87400.0, with half of the transactions that are smaller or equal to 2100.0

	Value	Type	Description
First Quartile	135.9	Float	Bigger than one fourth of the amounts
Median	2100.0	Float	Bigger than half of the amounts
Third Quartile	6800.0	Float	Bigger than three fouth of the amounts
Maximum Amount	87400.0	Float	Maximum amount
Mean	5924.15	Float	Mean
Number of accounts	4500.0	Integer	Number of distinct accounts
Number of transactions	40400.0	Integer	Number of transactions

Table 3.1: Some statistics about the Berka set

3.1.2 The PROMETEIA dataset

PROMETEIA dataset is private data that is provided by PROMETEIA company. It contains more than a million transactions, spread on 9841 cards and divided into two categories, debit/credit transactions. The advantage of this dataset is the age, from 2018 to 2020, and then it reflects the behavior of customers in a contemporary bank in Italy.
There are six features in the data, let mention all of them below:

- ID_CARTA: Card's number
- DT_TRANS: Transaction date
- CD_VALUTA: Currency of the transaction
- IMPORTO: Transaction amount
- FLAG_DARE-_AVERE: Transaction type
- MCC: Merchant Category Code

In addition to the features described in the previous dataset, there is the card number column, since the data include bank cards and not every kind of transaction. And Merchant Category Code (MCC) column is a four-digit number listed in ISO 18245 for retail financial services. An MCC is

used to classify a business by the types of goods or services it provides, see [**MCC list**].
Here are some statistics about the amount feature. The maximum amount is 500000.0. Half of the transactions are smaller or equal to 21.0 EUR. we will see more details in the next section 3.2.

	Value	Type	Description
First Quartile	8.63	Float	Bigger than one fouth of the amounts
Median	21.32	Float	Bigger than half of the amounts
Third Quartile	65.44	Float	Bigger than three fouth of the amounts
Maximum Amount	500000.0	Float	Maximum amount
Mean	101.814	Float	Mean
Number of accounts	9841.0	Integer	Number of distinct accounts
Number of transactions	50309.0	Integer	Number of transactions

Table 3.2: Some statistics about the PROMETEIA set

3.2 Data analysis

Data analysis is a process of investigating data to deduce information that can help in deciding on a business, confirming theories, or refuting existing models in science for example. Data analysis can also be exploring large sets of data using sophisticated software to identify undiscovered patterns and establish hidden relationships.
Data cleansing is the first important step. It is a process aimed at identifying and correcting corrupted, inaccurate or irrelevant data. This fundamental step in data processing improves the consistency, reliability, and value of data. The most common causes of data inaccuracies are missing values, entries that do not appear in the correct location, and typos. In some cases, data cleaning requires certain values to be entered or corrected; in other cases, the values should be simply deleted.
Data Visualization is an important part of data analysis. It can be defined as a visual and interactive exploration of the data and its graphic representation. Visualization is playing an increasingly important role in helping us make sense of the billions of rows of data generated every day, trends can be perceived quickly and easily. It helps to observe things in a very accessible way. Visualization facilitates the transmission of information in a universal and makes it easy to share ideas with others.

3.2.1 Data exploration and visualization: Berka dataset

Plot distribution of the transaction amount

The histogram presented in Figure 3.1 shows the probability distribution of the transaction amount of all the Berka dataset with a logarithmic scale. A general trend shows that the number of transactions is conversely proportional to the transaction's amount. Both types of debit and credit transactions are combined because they follow the same behavior. The histogram shows, in particular, the fact that the more the amount is big, the fewer are the owners and dealers of such amount.

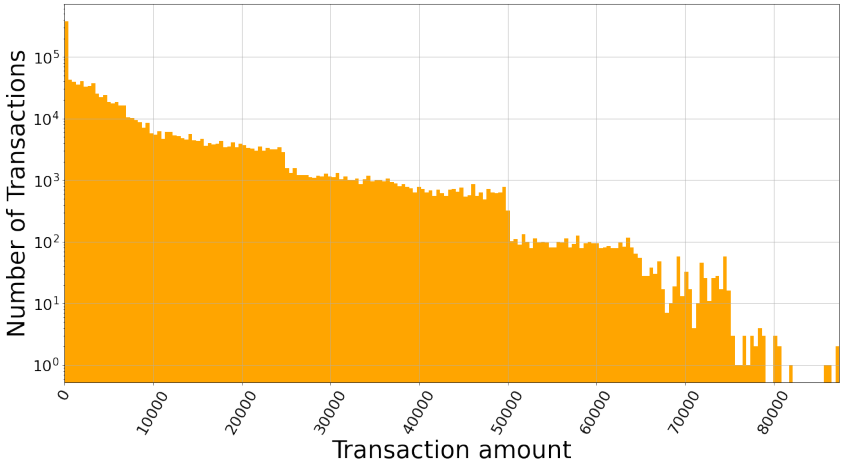

Figure 3.1: PDF of the amount for Berka dataset

However, the large size of the data in both axes and its leaning to the left does not allow us to check the details. Figure 3.2 shows a clip into the histogram between 0 and 5000 CZK.
Histogram clipping is zooming in an interval where the outside values are counted with the interval edge's values.
It shows peaks in some round numbers, multiple of 100 like 300, 400, etc. These peaks reach nearly $< 10^4$ customers. The explanation is that most people use a round number in their transactions if the amount is not a price or bills.

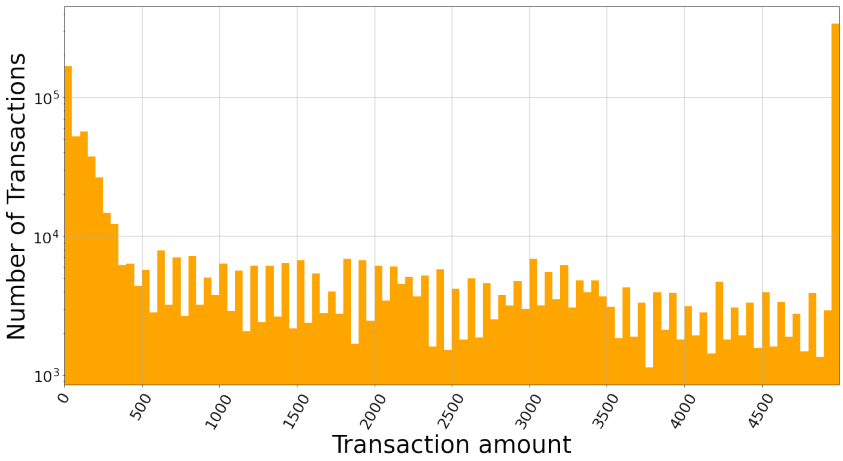

Figure 3.2: PDF of the amount for Berka dataset between 0 to 5000 CZK.

Another observation is that small transactions occur much more often than big ones. This is why we clip the histogram between 200 and 5000 CZK in figure 3.3.
The two edges of the histogram have the same height, which means that the sum of transaction amounts below 200 CZK (40% 340, 000) is comparable to the count of transaction amounts larger than 5000 CZK (up to 90,000 CZK).

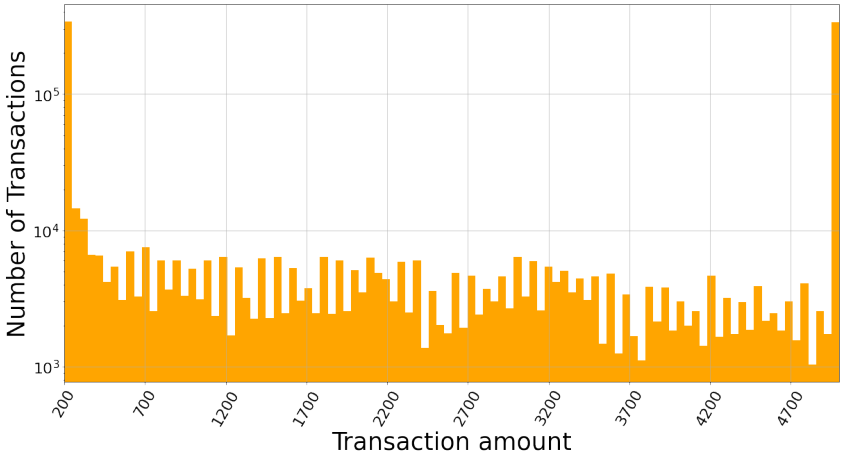

Figure 3.3: PDF of the amount for Berka dataset between 200 to 5000 CZK.

Plot distribution of the number of transactions per month

Figure 3.4 shows the probability distribution function of the number of transactions per "client-month" pair (the x axis is "number of transactions", the y axis is "number of client-month pairs with x transactions for that client in that month". We observe that most clients have around five transactions per month. We use such observation in targeting a category of users for advertising a bank product, predicting transaction outcomes, or detecting strange behavior in the transactions.

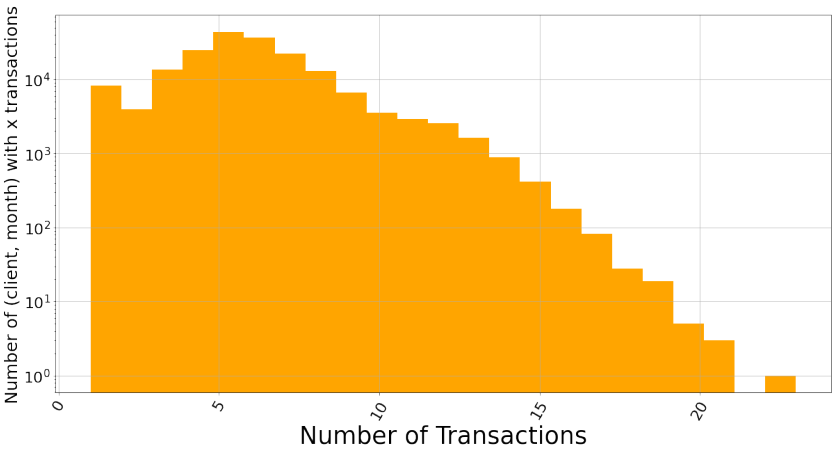

Figure 3.4: PDF of the transactions per client/month for Berka dataset

3.2.2 Data exploration and visualization: PROMETEIA dataset

Plot distribution of the transaction amount

The histogram presented in Figure 3.5 shows the probability distribution of the transaction amount for the PROMETEIA dataset on a logarithmic scale. An interesting observation is that most of the transactions are below 30,000 EUR and there are a few large transactions up to 500,000 EUR, where two of these transactions $230,000.0$ and $500,000.0$ are withdrawal from the same card, while the two others are from two different cards.

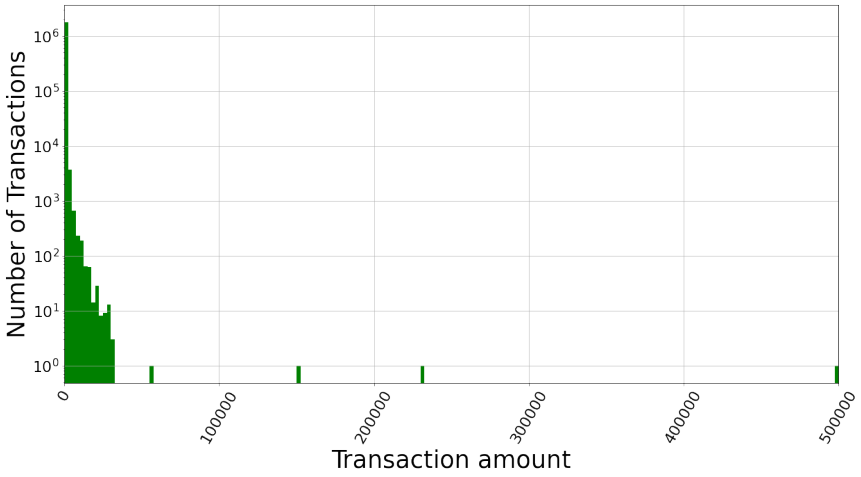

Figure 3.5: PDF of the amount for PROMETEIA dataset

Figure 3.6 is a zoom into the range between 0 and 1000 EUR of the data. Regardless of the peaks in round numbers, the histogram shows a hyperbolic trend; the number of transactions decreases strongly with respect to the amount of the transaction. The peaks in some round numbers can have the same explanation as for the Berka dataset, that is most people use round numbers in their transactions.

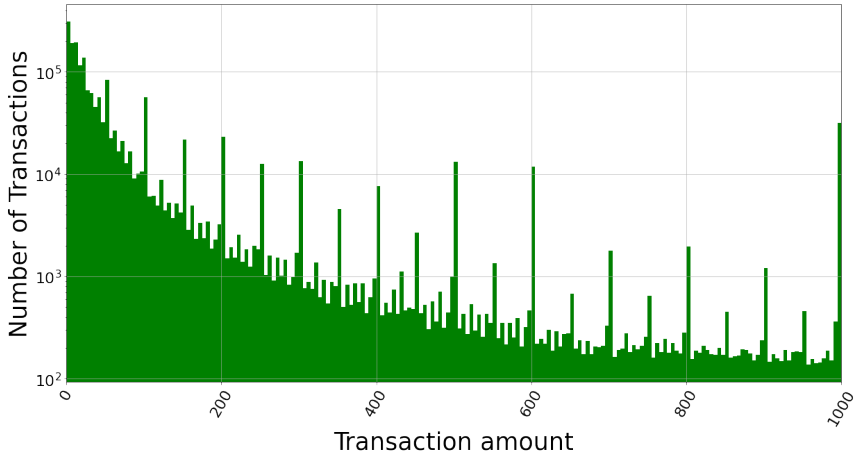

Figure 3.6: PDF of the amount for PROMETEIA dataset between 0 to 1000 EUR.

The high number of transactions below 50 EUR can be explained by looking at the MCC of these payments.

MCC The MCC gives information about the nature of the transaction, food, clothes, cars, health ... There is a list in the Citibank website that gives to each range of number a category of business. For example, after looking at the MCC of the payments below 50 EUR, we see that 8.7% of the transactions below 50 EUR have an MCC equal to 5411 which is the one related to grocery stores.

Plot distribution of transactions per month

Figure 3.7 shows the distribution of the number of transactions per client per month with a zoom between 0 to 130 transactions.
First, there is a difference between Berka data in the number of transactions per month, it is because nowadays, bank cards are more commonly used for daily payments that are small amounts (less than 100 EUR) which is observable in figure 3.5.
The number of cards is inversely proportional to the number of payments/month which is normal because it is rare for a common person to have more than 5 transactions per day and 150 per month which is shown in the zoomed area. However, we observe few cards that are having a very high number of transactions per month. This illustrates how such investigation can be useful in the detection of anomalies or this particular case fraud.

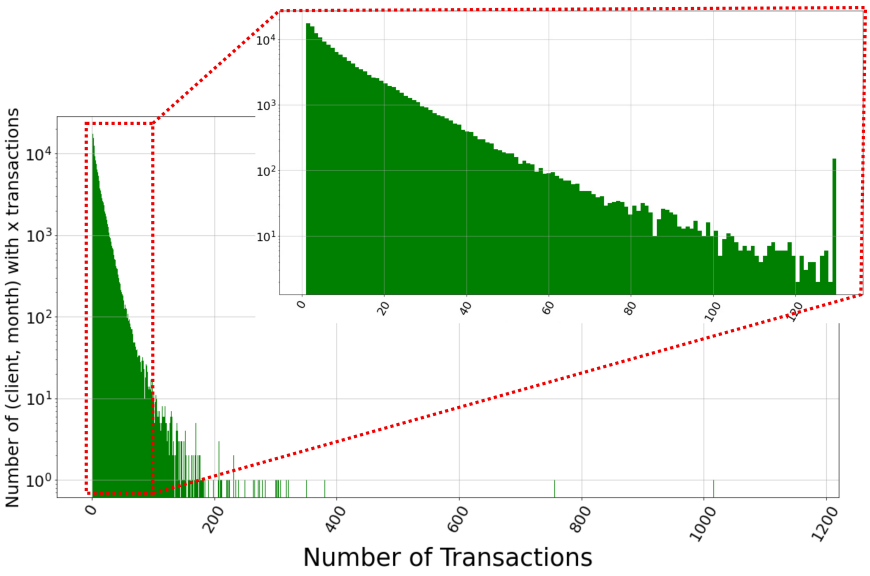

Figure 3.7: PDF of the transactions per client/month for PROMETEIA dataset

Chapter 4

Rule-based strategy for the detection of recurring transactions

A rule-based strategy is a logical program that uses human-defined rules to make deductions and choices to perform automated or semi-automated actions. Automating complex, lengthy, manual processes dramatically reduces the time and cost of critical tasks such as data validation, cleansing, integration, and enrichment. Even specialized tasks such as searching specific patterns or improving the accuracy of previous work can be automated with dramatic results. The power and flexibility of the rules engine mean that complex or subtle decision-making does not have to be sacrificed to achieve automation. The rule-based approach makes it possible to encode these decisions in the form of rules and therefore to automate them. The main advantage of a rule-based strategy is the consistency it provides since the same treatments must be applied to the same instances. Disadvantages are that the rules rely on domain experts (expensive/hard to find), and domain expertise is not forever, so maintenance of rule-based systems is hard.
Moreover, in this study, it is not easy to choose the right balance between strictness and laxity. If we are too strict and make the rules strict we miss cases or select the wrong one because of their laxity. . For example, if we choose the minimum number of occurrences too high like six, we can miss payments with a high amount that occurs five times. On the other side, if we choose a small number like three, the algorithms can include payments like 9.99 EUR, which are common payments.

4.1 Applications and relevance of detection of recurring transactions

Why it is important to know if the account has recurring transactions, there several motivations that we will list below.

Risk of stopping payments and loans For granting a loan to a client or in case of credit card usage, the bank needs to know how stable and regular are the customer's transactions, to guarantee the return of their money. It takes into account regular income and expenses to assess the likelihood of the loan being paid back in the future. The sudden stop of a regular source of income can be used as a signal to trigger risk management actions such as renegotiation. The bank needs to have an estimation of how many customers are eligible to request a loan, how many of them are probable to ask that, how much money all these requests need, and all are based on the income and sustainability of the customer's account.

Actual income and investment capability There is a law that obliges the banks to evaluate the client's sustainability. So, companies like banks need to know the incomes of customers and their capabilities to buy something or make a loan for buying it. This can help banks to categorize the customer's advertisements as well. The other thing would be the level of service that the bank provides for different customers, which is not the same: customers with higher deposits would get more facilities/priorities in the services, in order to encourage them to keep investing in that bank. For example, banks provide health insurance for customers that have deposits higher than a specific amount. Also, different features and roofs of credits will be considered in their cards.

Anti-money laundering Banks are ordered to detect illegal money that is transferred to or from suspicious sources/receivers. For insurance companies, knowing some repeating issues regularly for some customers is important, as it may mean that they want to take advantage of their insurance coverage. For example, making artificial cases of burning their insured house.

4.2 Rule-based algorithm and implementation

We would like to define rules that will generate the algorithm for recurring transaction detection. Rules that will be at the base of our algorithm will be described in the next paragraph. The following plot shows one bank account's transactions. We can observe in the period 1994-1996 and the amount between 100 and 150 CZK the kind of payment we like to detect.

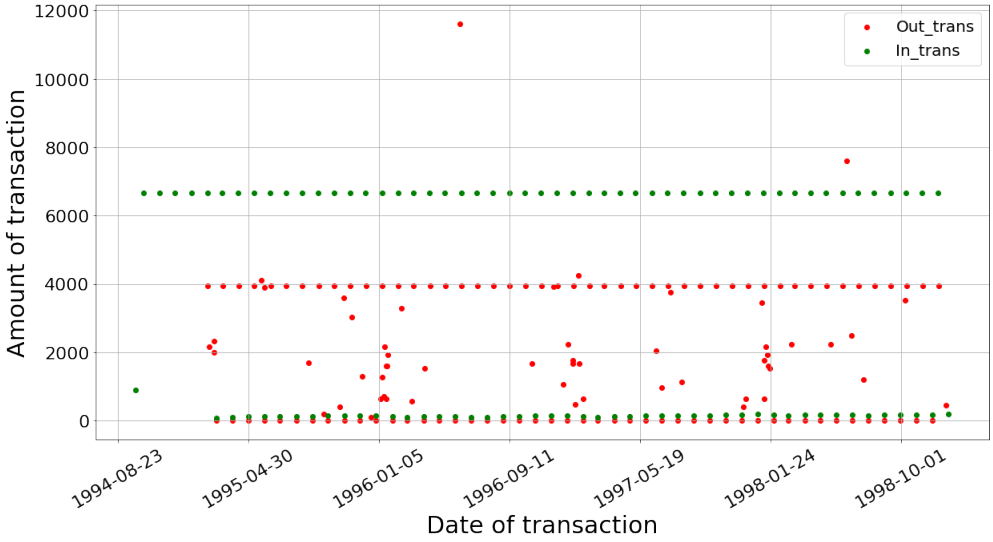

Figure 4.1: Transactions of account no. 6 in Berka data

Definition of a recurring transaction Recurring transactions are multiple ones made more or less regularly over time, with almost the same amount provided or received over some time.
Two things are difficult to formalize "more or less regularly over time" and "almost the same amount". The diagram in figure 4.2 illustrates our rule-based algorithm.

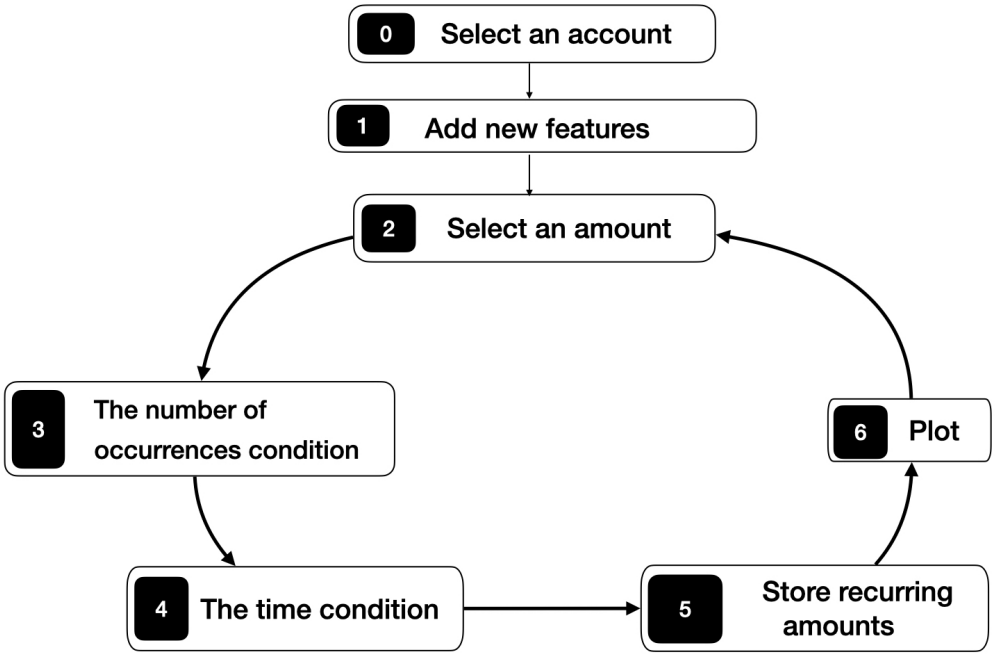

Figure 4.2: Rule-based algorithm's diagram

The algorithm works as follows. First, an account table is given as input. New features are extracted from the existing ones, regarding the time, as the day or the business day of the month, and the number of days or business days to the end of the month.
For example, if we look at the business days from "2020/11/26" to the end of November 2020, we have "2020/11/26", "2020/11/27", and "2020/11/30". The number of business days to the end of the month is three including the last day of the month. Such features are saved as new columns in the account table.
Another important feature is the number of occurrences of each amount with a defined tolerance, called "tol" (i.e. 1%). It means we count how many times it and/or its neighbors, occur and we save that value in a column called "freq" for frequency.
The neighbors of an amount x_0 are amounts that belong to the interval

$$Neighbors_{x_0} = [x_0 - tol * x_0, x_0 + tol * x_0].$$

After adding the needed features, let's check the conditions shown in the diagram. For each amount in the account table, we check the number of occurrences condition using the "freq" column. The condition can be "freq bigger than 5".
Let consider that x_0 is an amount that passed the first condition. In the second one, we check the time features of the occurrences of x_0 and the ones of all the occurrences of its neighbors. We check if all occurrences of the amounts that belong to $Neighbors_{x_0}$ are happening in:

- roughly same date of the month,
- or roughly same business day of the month,
- or roughly same number of days to the end of the month,
- or roughly the same number of business days to the end of the month.

If yes, then the amount x_0 is recurring.
There are two different implementations of the rule-based algorithm for the detection of recurring transactions. The difference starts at step 2 of the diagram 4.2, the first implementation uses Pandas' functionality, while, the second one uses a "for-loop" over the set of the transactions amounts. We can see both implementations in the module "detect_recurrency" 6.1 of the *bankdatainvestigation* package. We did run the first algorithm on 1000 accounts from each dataset. Some output examples are presented in section 4.4.

4.3 PySpark implementation: Pandas UDF with Apache Arrow

Pandas-UDF A user-defined function (UDF) is a Python function that can be executed row-wise on a PySpark DataFrame. UDFs allow for arbitrary Python code to execute scalably via PySpark but come at the cost of great serialization time because rows have to be moved to the Python runtime.
A Pandas UDFs serialize batches of rows as Pandas DataFrames using Apache Arrow to transfer the data and the job, speeding up serialization, and then within the Python runtime, it is possible to achieve further speedup using pandas and NumPy in a vectorized fashion instead of working on single rows.

Apache Arrow is a Python API for in-memory analytics that is in a server's random access memory (RAM). It contains a set of technologies that enable big data systems to process and move data fast between any computer languages [**9**].

Pandas-UDF in the detection of recurring transactions Since we have a Pandas implementation for the detection of recurring amounts, we take advantage of Pandas-UDF. In other words, the implementation of the functions remains the same but the signature of the function and the way we call it are different. That is because we need to make an external function that takes those arguments that do not need to be distributed by PySpark, and the internal function that will be distributed over different processors as shown in figure 4.3:

```
def DetectRecurrency(
                      amount_tolerance : float=0.01,
                      period_tolerance : int=6,
                      n_days : int=3,
                      client_col: str =None,
                      time_col: str = None,
                      amount_col: str = None,
                      config: dict = None
                      ):


    if config is not None:
        client_col = config.customer_id
        amount_col = config.trans_amount
        time_col = config.trans_date

    #Creation of the new columns
    def rb(trans_data : pd.DataFrame):

```

Figure 4.3: Signature of the function DetectionRecurrency using Pandas-UDF

The external function with specified arguments will be given to a PySpark command using "applyInPandas" as shown in figure 4.4 :

```
account_table = account_table.groupby('account_id').applyInPandas(rb, schema=
     "account_id long, trans_amount double, trans_date timestamp, freq double, rec boolean")
```

```
account_table = account_table.groupby(client_col).applyInPandas(rb, schema=
     "ID_CARTA long, IMPORTO double, DT_TRANS timestamp, freq double, rec boolean")
```

```
account_table.show()
```

Figure 4.4: PySpark command for calling the Pandas-UDF DetectionRecurrency function

4.4 Case studies

.
We show some results of the rule-based algorithm applied on both datasets. We choose to plot the output of "DetectRecurrencyI" that is easier to have a global view of the accounts and their transactions.

The Berka dataset Figure 4.5 showing the transactions of an example account number 6 together with the results of the function "DetectRecurrencyI". The algorithm could find three recurring transactions. Zooming into the transaction amounts between 0 and 300 CZK (shown on top of Figure), we better realize the performance of the function which can discriminate recurring values from noises, even if they are very close to each other.

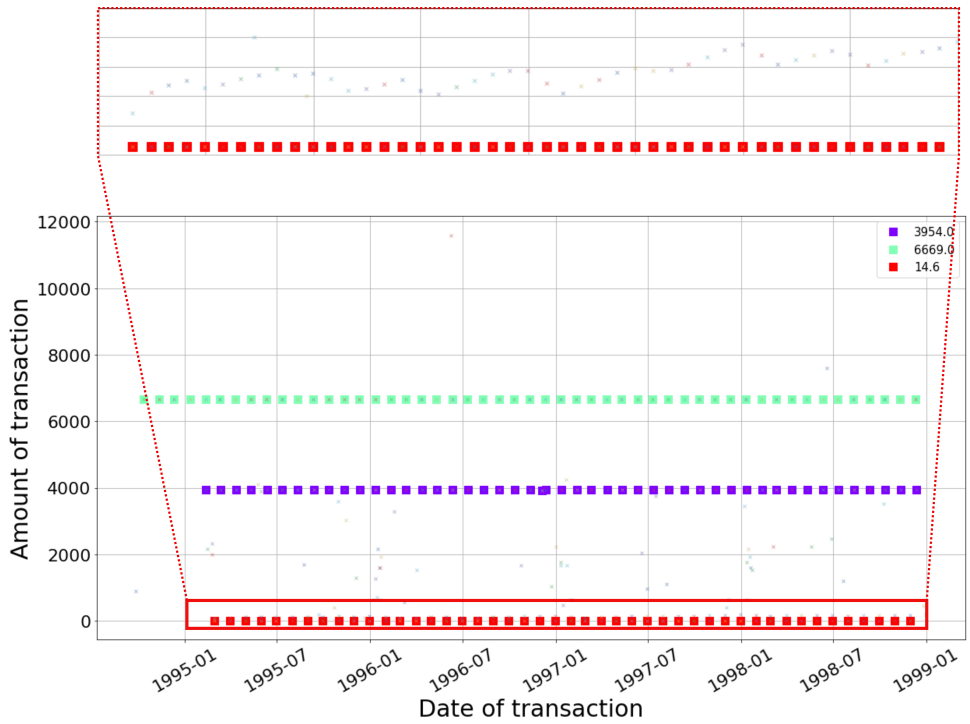

Figure 4.5: Recurring transactions of account no. 6 in Berka dataset using "DetectRecurrencyI"

Another example output of our algorithm is shown in figure 4.6 where four clusters are detected. However, noises are also visible in the zoomed plot on top of them between 14.6 and 297 CZK.

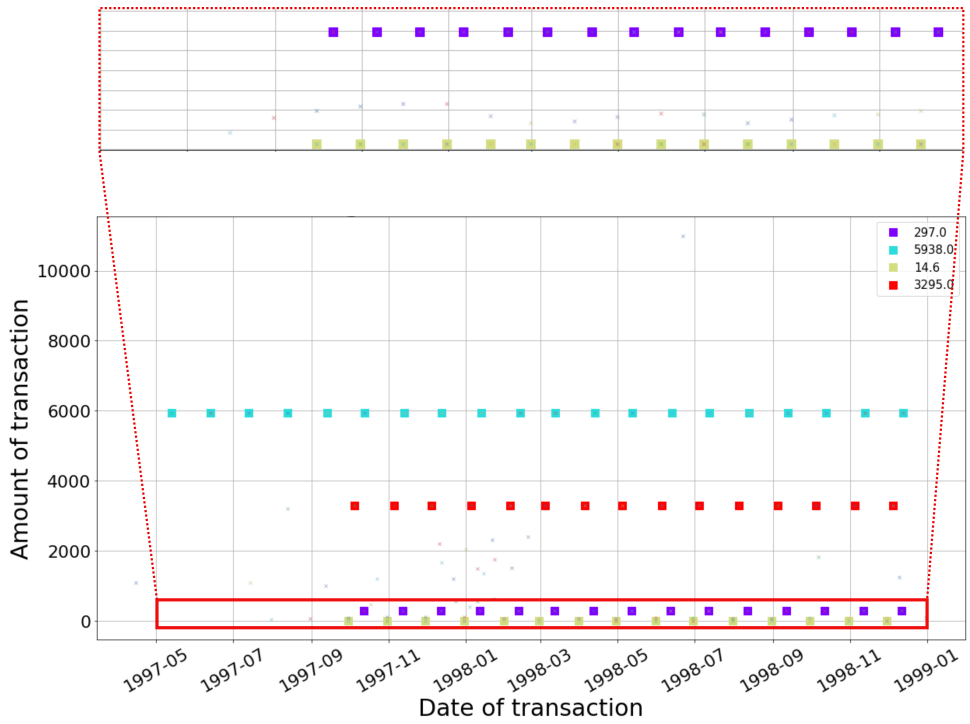

Figure 4.6: Recurring transactions of account no. 12 in Berka dataset using "DetectRecurrencyI"

PROMETEIA dataset We can observe in 4.7 only one detected recurring behavior which is a false positive since there is no total regularity in time, the payments are happening in the same period of the month but not every month.

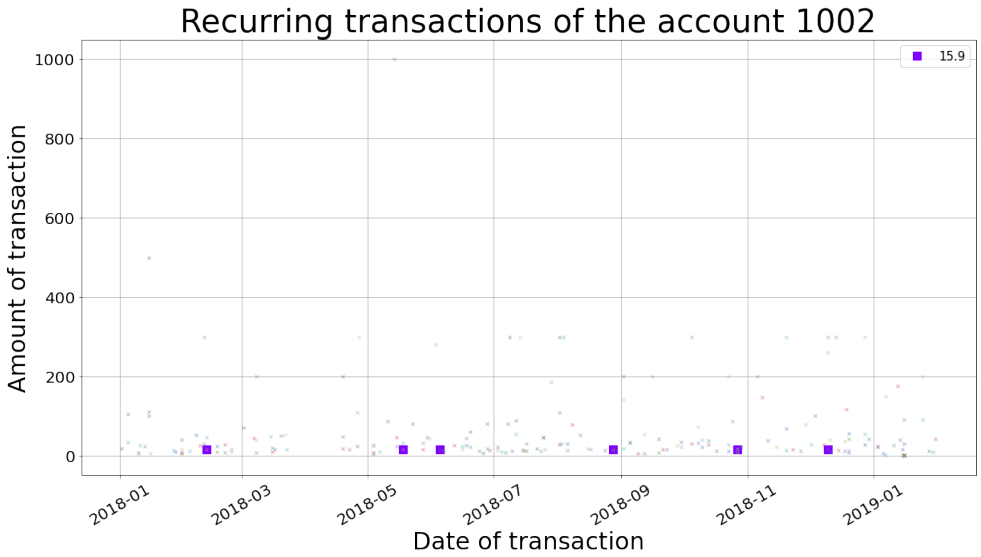

Figure 4.7: Recurring transactions of account no. 1002 in PROMETEIA dataset using "DetectRecurrencyI"

Chapter 5

Clustering

A cluster is a set of elements that are distinct from the others. Each element of a cluster has strong similarities with the other elements of the same cluster and must be different from the elements of the other clusters. So there is an idea of looking for distinct groups.
Clustering methods are unsupervised algorithms that allow to generate and find natural classes. For example, such a data mining method is used in marketing to discover the profile of certain groups of customers, and thus adapt to a market. The clustering method must be reliable, be able to create distinct clusters, be weakly sensitive to noise, and discover hidden patterns.

DBSCAN and detection of recurring transactions
Density-Based Spatial Clustering of Applications with Noise (DBSCAN) is a clustering algorithm whose goal is to discover clusters and noise in a dataset. It needs two arguments: the size of a neighborhood ϵ, and the minimum number of points MinPts. For each point, it calculates its ϵ-neighborhood, then, if this neighborhood contains more than the MinPts points, it does the same for each of them, and so on, until it can no longer expand the cluster. If the considered point is not an interior point, it means that it does not have enough neighbors, then it will be labeled as noise. This allows DBSCAN to be robust to outliers since this mechanism isolates them.

5.1 Projection on a cylinder

In this chapter, we want to detect transactions that occur regularly every month. We will use the DBSCAN algorithm on two axes, the amount ax and the day of the month ax. It will be applied on the "transformed" axes (see Figure 5.1). Let us consider the two features from our account table. The first is the amount of the transaction that is just a float number on a real axis. The second feature is the time of the transaction, that we project on the day of the month, for example, the first day, 15th day, etc. Since the days of the month are periodic, or almost periodic because not all the months have the same number of days, they can be represented as points on a circle.

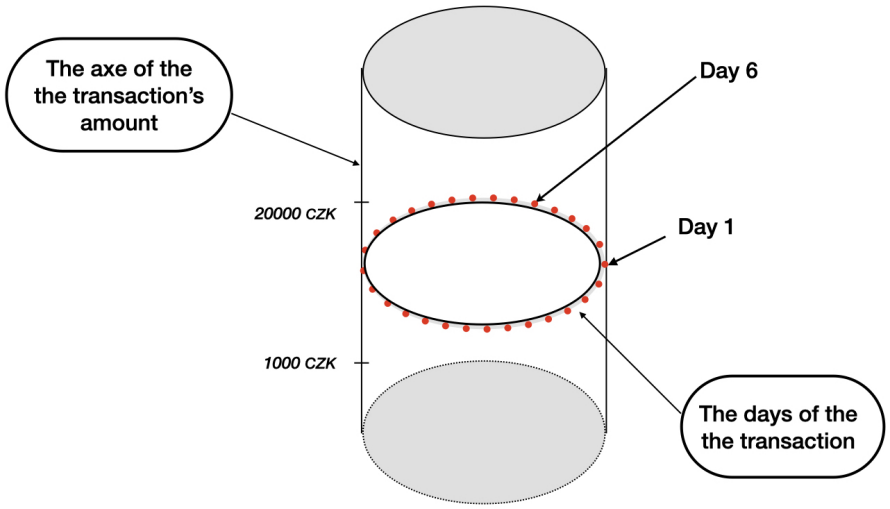

Figure 5.1: Cylindre projection of an account table.

The two coordinates of the circle are the sine and cosine of the day of the month, that we multiply to a number to make them in the same range as the amount ax. On the other hand, to catch up with the variability of the tolerance argument regarding the amount column, we apply a logarithm in base b where b is the suitable percentage of tolerance. For example, if we want to make tolerance of 1% in the amount we apply the logarithm with base 1.01 to the amount column.

These transformations on the dataset, allow us to choose the parameters in a meaningful way.
However, to complete the work, it would be suitable to check the selected fibers by DBSCAN algorithm. This is to have the periodicity in the recurring transactions. For example, this algorithm detects transactions occurring on the fifth of every month or every three months. Also, it can be done by adding a fourth dimension to the algorithm that is the month of the year.
This will be added after the defense.

5.2 Case studies

After the mentioned operations in the previous section, our two columns of data have now became three, the logarithm of the amount, sine, and cosine of the day of the month. We run the DBSCAN algorithm with $\epsilon = 2$, and $MinPts = 8$.
Some examples are presented below.
Figure 5.2, shows the result of the DBSCAN algorithm on account no. 1000 in Berka dataset with the cylinder projection. The gray points are noises, which means they do not belong to any cluster.

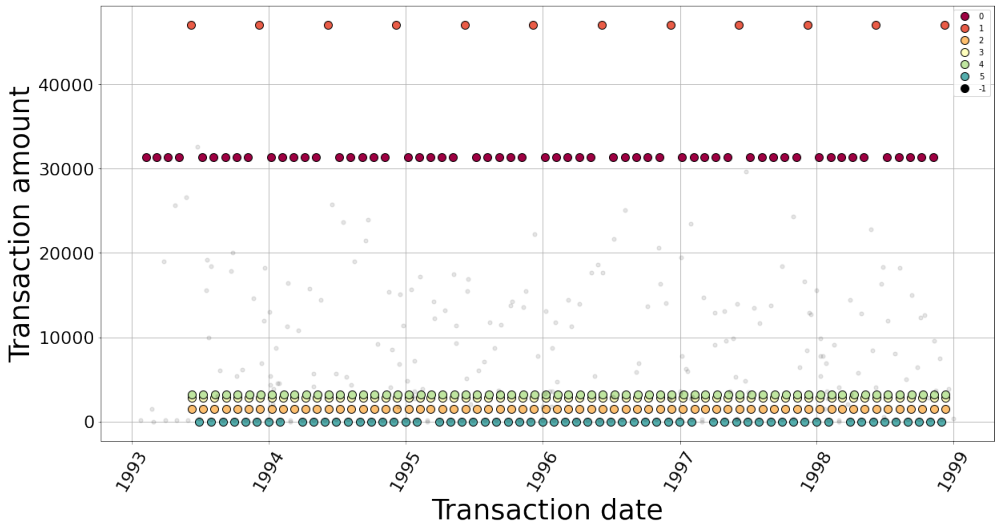

Figure 5.2: Result of DBSCAN on account table no. 1000 in Berka dataset in two dimensions.

Figure 5.3, shows the same account information but in a 3D view. We can see that cluster 0 became one point in the 3D representation because the

occurrences of the corresponding amount occurred on the same day of the month.
However, the occurrences of the fifth cluster are happening in three consecutive (regardless of the month and year) days because, in reality, they are happening on the last day of different months, which are 28, 30, or 31, which explains three dots.

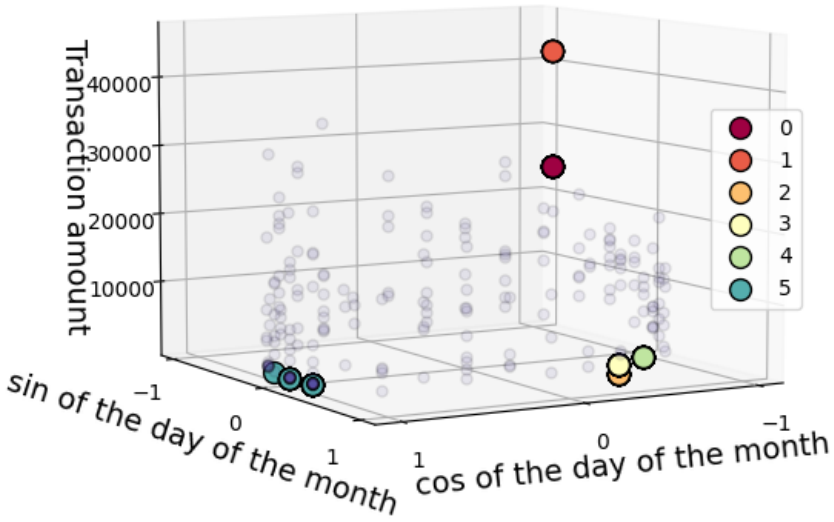

Figure 5.3: Result of DBSCAN on account table no. 1000 in Berka dataset in three dimensions.

Figures 5.4 and 5.5 show the output of DBSCAN algorithm with the same parameters as for the previous account $\epsilon = 2$, and $MinPts = 8$ on account no. 6 and no. 12 of the Berka dataset. These are the same accounts shown in section 4.4. We can see the same clusters using DBSCAN and the rule-based algorithm shown in section 4.4. Again, we observe the same cluster of transactions in different days which are last days of different months

(Figures 5.4 and 5.5).

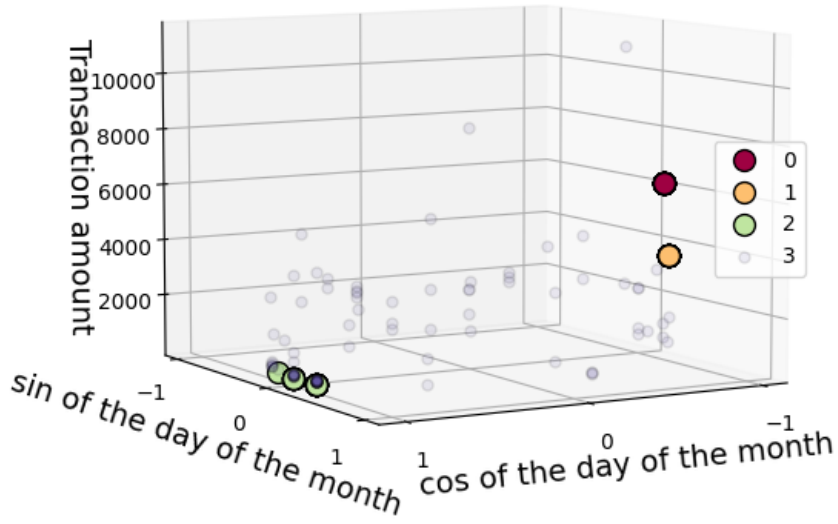

Figure 5.4: Result of DBSCAN on account table no. 6 in Berka dataset in three dimensions.

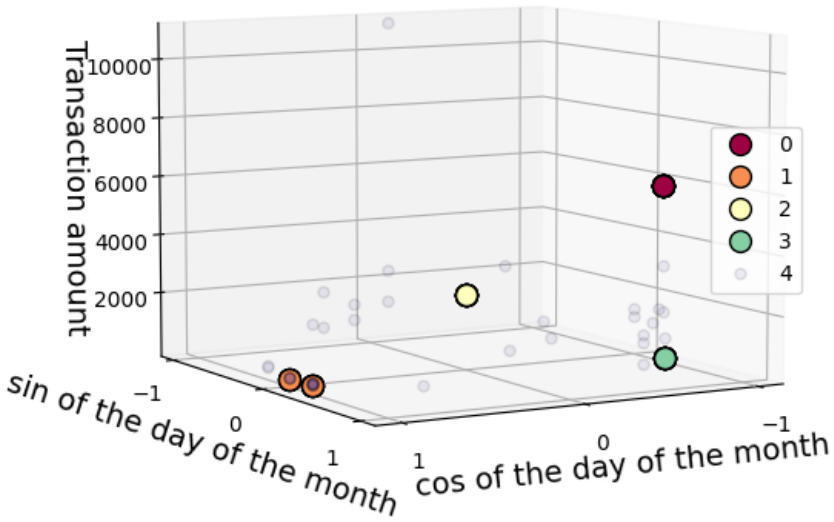

Figure 5.5: Result of DBSCAN on account table no. 12 in Berka dataset in three dimensions.

Chapter 6

PyPI Package: bankdatainvestigation

In the two previous chapters, we investigated both datasets and developed algorithms for the detection of recurring behaviors. Here, we create a package that provides an implementation of all mentioned algorithms on any other datasets. The package is uploaded on the 'Python Packages Index' (PyPI) which is a repository of software for the Python programming language. We provide this package with a free license, having the goal of developing it in near future.

6.1 bankdatainvestigation package

bankdatainvestigation is a Python package to investigate Bank data using Pandas, Matplolib libraries, and Pyspark framework. It contains four sub-packages named "Customer", "PySpark-implementation", "pdf" and the configuration sub-package "config-load".
In the next paragraph, we will describe all the functions implemented in the package *bankdatainvestigation*. All the public functions take as argument the DataFrame we are looking at, and the names of columns that the function is concerned by, or the name of configuration class in the case of our data, that is listed in the configuration sub-package.

Customer The "Customer" sub-package contains two modules "customer.py" and "detect-recurrency.py".

"customer.py" module The module "customer.py" contains the implementation of three functions :

1. The first one gives the number of customers in the data named "number-of-customers". It takes two arguments, the DataFrame and the column name (that depends on the data) or the name of the configuration class in the case of our data.

2. The second function gives the number of transactions in the data, named "number-of-transactions". It takes two arguments, the DataFrame and the column name (that depends on the data) or the name of the configuration class in the case of our data.

3. The last one, compute the number of active accounts for a given year and/or month. A client is active in a period if it has at least one transaction in that period. The function is named "number-of-active-accounts". If the user does not specify the year and/or month, the function will return the number of active accounts for all the available periods. It takes six arguments, the DataFrame, and the names of three columns or the name of the configuration class in the case of our data.

"detect_recurrency.py" module The module "detect_recurrency.py" contains the implementation of two public and three private functions. The public ones are "DetectRecurrencyI" and "DetectRecurrencyII" and their take nine arguments listed below :

- trans_data: type Pandas DataFrame. It is the DataFrame of the chosen customer,
- amount_tolerance: type float set by default to 0.01. the radius of the interval centered in an amount x_0,
- period_tolerance: an integer type set to six by default, It is the minimum number of the accuracy
- n_days: an integer type set to three by default, the accepted variance in the payment day.
- client_col: a string set to "None", the name of the client column in the case of new data,
- time_col: a string set to "None", the name of the time column in the case of new data,
- amount_col: a string set to "None", the name of the amount column in the case of new data,

- config: a Python class set to "None", in the case of Berka or PROMETEIA data,
- plot: a boolean parameter. If set to "True", both functions can return a plot scatter of all the transactions where the noise values are in grey and the considered as recurring values in different colors. for plotting or no the output

The private functions in the module "detect_recurrency.py" are :

- "_complete_table" function, it returns the same table with additional time columns that are needed by the functions "DetectRecurrencyI" and "DetectRecurrencyII" for verifying the time condition.
- "_detect_recurrency" function, it is used by "DetectRecurrencyII" and it takes a DataFrame as an argument and returns a list and dictionary. The list contains couples, each is a recurring amount and its frequency in the DataFrame. The keys of the dictionary returned by the function "_detect_recurrency", are the recurring amounts and values are the corresponding DataFrame.

PySpark implementation The "PySpark_implementation" sub-package contains the PySpark implementation of "detect_recurrency" function as detailled in the 4.3. It is a Pandas-UDF function that will be called using the "applyinPandas" command on PySpark DataFrame.

pdf The "pdf" sub-package contains the following functions :

- PDF_amount_transaction function, that plots the probability distribution function of the transaction amount with respect to two optional parameters (client_ID and year). An output of PDF_amount_transaction function is shown in Figure 3.1. The x-axis contains the transaction amount and the y-axis is the number of a client with the x amount. It takes as arguments the DataFrame, three-column names, or the name of the corresponding configuration class in the case of our data. Also, we can specify the range of amount by clipping the year we want to consider, the bins, and the logarithmic scale that is set as "True" by default.
- PDF_transactions_client_month that plots the probability distribution function of transaction per "client-month" pair. The x-axis gives the number of transactions and the y-axis number of client-month pairs

with x transactions for that client in a month. It takes as arguments the DataFrame, column name, or the name of the corresponding configuration class in the case of our data. It also accepts the same tuning parameters to the PDF_amount_transaction function.

config-load module The configuration sub-package named "config-load", is useful for our bank datasets. It allows us to use the same column names for both datasets just by importing and using the right class.

Installation and importing of the package The *bankdatainvestigation* package, is shared on PyPI. The following command downloads and installs the version "1.0.0" of this package :
- pip3 install bankdatainvestigation==1.0.0 This version works with Pandas==1.2.0, Numpy==1.19.5, PySpark==3.0.1, Pyarrow==2.0.0, and Matplotlib==3.3.3. To install Pyspark, one needs to install the appropriate Java (version 8 is tested). There an attached file "requirements.txt" containing all packages used in the work environment of this package.
After installing the package, the following lines will import it in Python:
- import bankdatainvestigation.Customer.customer as cs
- import bankdatainvestigation.Customer.detect-recurrency as dr
- import bankdatainvestigation.spark-implementation.detect-recurrency as drs
- import bankdatainvestigation.pdf.pdf as pdf

Chapter 7

Conclusion

In this project, different methods are investigated to detect the recurring transactions in bank data. Then, the main ones are gathered as a package called bankdatainvestigation that is proposed in chapter 6.
We were able to analyze the main features of the two studies' datasets of Berka and PROMETEIA. We detected recurring transactions of the clients of both datasets. We could see more recurring transactions in account data (Berka) than card data (PROMETEIA), which is normal since the card is mostly used for small transactions that we observed in chapter 3. The card data have a lot of transactions but without specific patterns for high amounts like recurrency. However, for small amounts, we could see payments related to cellphone bills, Netflix, etc.
We also transformed our data so that recurrency patterns could be recognized using classical unsupervised learning techniques.
Software like PySpark shows its importance when we face big datasets in scales of terabyte or petabyte. However, the main goal of this project was to learn these tools and be able to implement them later on big data.

Bibliography

[1] https://pandas.pydata.org/docs/

[2] https://matplotlib.org/stable/contents.html

[3] https://scikit-learn.org/stable/user_guide.html

[4] https://docs.conda.io/en/latest/

[5] https://spark.apache.org/docs/latest/sql-pyspark-pandas-with-arrow.html

[6] https://spark.apache.org/docs/latest/api/python/

[7] https://dlab.epfl.ch/2017-09-30-what-i-learned-from-processing-big-data-with-spark/

[8] https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-DataFrames-and-datasets.html

[9] https://medium.com/datadriveninvestor/use-apache-arrow-to-assist-pyspark-in-data-processing-6c1cce134306

[10] https://medium.com/walmartglobaltech/decoding-memory-in-spark-parameters-that-are-often-confused-c11be7488a24

[11] https://www.dm.usda.gov/procurement/card/card_x/mcc.pdf

[12] https://www.py4j.org

[13] https://www.citibank.com/tts/solutions/commercial-cards/assets/docs/govt/Merchant-Category-Codes.pdf

[14] https://www.jpmorganchase.com/content/dam/jpmc/jpmorgan-chase-and-co/institute/pdf/institute-estimating-family-income-report.pdf

å