INTRODUCTION TO BLACKLIST MATCHING PLATFORM FOR FINANCIAL ENTITIES

GREY SYSTEMS

Versión 1.0.0

1. Introduction

A *blacklist* in financial terms is *'A list of people, organizations, or countries that are disapproved of and that people avoid doing business with, for example because they are dishonest'*. Deception is part of the 'natural' behaviour: it has been shown that even animals commit deceptions in order to obtain a profit of their own.

In the financial domain, the main reason for fraud is obtaining money with illegitimate purposes. As providers of technological platforms for the sector, we have detected the need to help our clients fight fraud. Therefore, we have started a line of research work whose purpose is the creation of instruments that help our clients fight the phenomenon of financial fraud.

This document is focused on one of the most demanded functionality in the financial sector, not only to fulfil legal requirements by governments but to reduce the risk and frauds, is to monitor and screen persons and entities against the different blacklist published by governments and private entities around the world.

The current nature of the actual published blacklists make the process of matching customers difficult and costly to financial entities for the following reasons:

(i) *Heterogeneity*: Blacklist are published by different entities without any standard format, nor standard data within registries, making it difficult to integrate different lists against current matching processes.

(ii) *Updates*: The blacklists can be thought of as living organisms, they grow over time, change etc. Keeping them up to date is essential to maintain the financial risks under control.

(iii) *Size*: A blacklist typically contains millions of records. The efficiency of simple matching algorithms (exact match, partial match) makes this operation costly in terms of performance, potentially reducing the number of times a person/entity can be matched against blacklists.

(iv) *Consolidation of data*: The collection and integration of data from multiple sources into a single blacklist composed of many of them.

(v) *Matching algorithms*: Last but not least, we have detected that most of the matching algorithms use either exact matching or partial matching but they ignore a simple fact; users are humans, we make mistakes when writing names and this situation is even worse when we write transliterate in a foreign language; an algoritm capable of detecting phonetical similarities is required.

2. The matching algorithm

The concept of near or inexact (*fuzzy*) matching is well established in the domains of information retrieval and computer science domain, where it is also known as 'approximate string matching' or 'string matching allowing errors'.

In essence, the near matching process comprises the identification of data items (text strings) which are not identical but may be permitted to differ up to a pre-set limit, which may for example be determined by the number of characters which are different (the *edit distance* approach) or by a computed similarity exceeding some designated threshold according to a preferred metric. An additional requirement might be the ability to distinguish between likely true versus false hits of the same measured distance or similarity, based on characterisation of real-world errors from the domain in question, where appropriate training data is available.

Probably the most widespread application of near matching is in the numerous spell checking applications developed either as standalone programs or as integrated apps into a range of word processing applications offering either spell checking of pre-entered text or, in some cases, checking text as it is being typed. However, such an approach is sub-optimal for purely data such as names of persons or entities, for the following reasons (among others):

(i) Standard reference dictionaries contain largely plain text terms (irrelevant in the present domain) and not person/entities names.

(ii) Algorithms for use with real person and entities names must be scalable so that they perform acceptably agains large reference dictionaries (the blacklists) containing millions of correctly names.

2.1. The Levensthein distance algorithm.

In information theory, linguistics and computer science, the *Levenshtein distance* is a metric for measuring the difference, or distance, between two sequences of characters. Informally, the Levenshtein distance between two words is the minimum number of single-character edits (insertions, deletions or substitutions) required to change one word into the other.

The Levenshtein distance may also be referred to as *edit distance*, although this term may also denote a larger family of distance metrics.

Date: 30 de noviembre de 2018.

Mathematically, the Levenshtein distance between two strings a and b (of length $|i|$ and $|j|$ respectively) is given by $\mathrm{lev}_{a,b}(|a|,|b|)$ where:

$$\mathrm{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j)=0 \\ r(a,b,i,j) & \text{otherwise} \end{cases} \tag{2.1}$$

$$r(a,b,i,j) = \min \begin{cases} \mathrm{lev}_{a,b}(i-1,j)+1 \\ \mathrm{lev}_{a,b}(i,j-1)+1 \\ \mathrm{lev}_{a,b}(i-1,j-1)+1_{(a_i \neq b_j)} \end{cases} \tag{2.2}$$

where $1_{(a_i \neq b_j)}$ is the indicator function equal to 0 when $a_i = b_j$ and equal to 1 otherwise, and $\mathrm{lev}_{a,b}(i,j)$ is the distance between the first i characters of a and the first j characters of b.

For example, the Levensthein distance between 'Juan Garcia' and 'Juan Gracia' is 3, since the following edits change on into the other, and there is no way to do it with fewer than three edits:

- Juan G**a**rcía → Juan G**r**acia (substitute *a* by *r*)
- Juan Gr**r**cía → Juan Gr**a**cia (substitute *r* by *a*)
- Juan Grac**í**a → Juan Grac**i**a (substitute *í* by *i*)

Here we can see some limitations of the Levensthein algorithm:

(i) Transpositions are detected as 2 changes. In our example the change 'García' to 'Gracía', while lexically that can be managed as a simple change: just transpose *ar* by *ra*.
(ii) Phonetically, *í* can be understood as *i*. For instance, that can be a common mistake on English speakers writing spanish names with diacritics.

2.2. **The Damerau–Levenshtein distance algorithm.** The *Damerau-Levenshtein* distance differs from the classical Levenshtein distance by including transpositions among its allowable operations in addition to the three classical single-character edit operations (insertions, deletions and substitutions).

It can be thought as an extension to *Levenshtein* and can be expressed with the function $d_{a,b}(i,j)$ whose value is a distance between i-symbol prefix (initial substring) of string a and a j-symbol prefix of string b:

$$d_{a,b}(i,j) = \min \begin{cases} 0 \\ \quad \text{if } i{=}j{=}0 \\ d_{a,b}(i-1,j)+1 \\ \quad \text{if } i{>}0 \\ d_{a,b}(i,j-1)+1 \\ \quad \text{if} j{>}0 \\ d_{a,b}(i-1,j-1)+1_{(a_i \neq b_j)} \\ \quad \text{if } i,j{>}0 \\ d_{a,b}(i-2,j-2)+1 \\ \quad \text{if } i,j{>}1 \text{ and } a[i]{=}b[j{-}1] \text{ and } a[i{-}1]{=}b[j] \end{cases} \tag{2.3}$$

Each recursive call matches one of the cases covered by the Damerau-Levenshtein distance:

(i) $d_{a,b}(i-1,j)+1$ corresponds to a deletion (from a to b).
(ii) $d_{a,b}(i,j-1)+1$ corresponds to an insertion (from a to b).
(iii) $d_{a,b}(i-1,j-1)+1_{(a_i \neq b_j)}$ corresponds to a match or mismatch, depending on whether the respective symbols are the same.
(iv) $d_{a,b}(i-2,j-2)+1$ corresponds to a transposition between two successive symbols.

Taking into account this new modification over original Levenshtein algorithm we can now check that the distance from 'Juan García' and 'Juan Gracia' is now of two edits:

- Juan G**ar**cía → Juan G**ra**cia (transpose *ar* with *ra*)
- Juan Grac**í**a → Juan Grac**i**a (substitute *í* with *i*)

2.3. **Transposition of Terms.** Although the Dameray-Levensthein algorithm provides us a way of determine how similar two names are, it's based on ordered strings, so if we compare 'Juan Garcia' and 'Garcia Juan', we will get a distance value of 9 which over a string size of 11. However, both names are pretty similar.

To solve this situation, instead of comparing the names like they are given, we can use a different approach, and sort name's terms before applying the algorithm, and that will result in a distance of 0.

So now, with this approach, we have an edit distance of 0. This approach, however, cannot be optimal for a customer, as the names are not equal, but pretty similar. The algorithm proposed by Grey Systems will generate two different scores for any comparison, with and or without transposition of the terms applied.

2.4. **Normalisation filters.** Through the previous subsections of the proposed matching algorithm we have now a well stablished way of measuring the similarity between two names, we could suffer from some predictable issues, due to the nature of how the names are mainly inserted in financial softwares. The users are usually in charge of inserting data, and they can introduce mistakes (transpositions, mispelled names, etc.), but also they could introduce unexpected characters, like double blank spaces, special characters that are not phonetically accepted for our comparision objectives, diacritics, etc.

Programs should always compare canonical-equivalent strings in order to properly determine comparision results, so both the data loaded from external lists and the data that comes from external resources, and the data that comes from users must be pre-processed before applying the similarity algorithm described above.

Due to the phonetic nature of person and entity names, the concept of normalisation filter must be added to algorithm so that we can customise the behaviour of the algorithm.

As an example of the different filters supported by the algorithm, we will have:

(i) Replace diacritics characters by its correspondent phonetic character. For example 'á' will be replaced with 'a'.
(ii) Remove non-phonetic characters, like `()-^`, etc.
(iii) Remove non required spaces (`trim()`).
(iv) Remove numbers.
(v) Use only uppercase characters.

With this normalisation filters, if we recover the previous example, the distance between 'Juan García', 'Juan Gracia' will be now 1, after the normalisation process as the strings will be computed as 'JUAN GARCIA' and 'JUAN GRACIA' respectively.

2.5. **Terms expansions.** Using the algorithm described above, we have a fine tuned algorithm to detect similarity between names, assuming we will always have the users to insert the full name in an ordered form such as `first_name middle_name last_name1 last_name2`. Unfortunately, this is not happening very frequently in some financial operations as transferring money to a beneficiary, since the sender does not know always his full name.

So, if we have a person blacklisted with the name 'JUAN ANTONIO GARCIA SANTOS', very common in spanish speaking countries, it's usual to know only the first lastname of a person, a sender can send money to 'JUAN ANTONIO GARCIA' resulting in a distance of 7 which compared to the length of the string 34 raises a high difference, which can result in a false matching, however the names are quite similar (3 of 4 terms match).

To accomplish that the algorithm will support what we have called `num_expansions`, a configurable parameter that can determine if the algorithm can perform recursions to search by the name terms separately. So:

(i) If full name does not raise positive results, then:
(ii) Try to search with again with $n-1$ terms.
(iii) If no results and $\max(exp) > \text{current}(exp)$ and $n-1$ terms are greater than 0, search again.

2.6. **Score result.** In order to make the things easier for consumers of the Grey Systems similar matching algorithm against blacklist, the algorithm will return an indicator telling the caller if the name given as input to the algorithm is matched against the different blacklist in the system and how much similar; in absolute terms; is the name to the found results in the blacklist.

2.6.1. *Similar scoring S_m.* The similar scoring between two names is calculated using the edit distance provided by the Levenshtein algorithm divided by the max length of the 2 names, raising a value between $0 < S_m < 1$, that can be expressed with the formula:

$$S_m(i) = \begin{cases} 0 & \text{if } results = 0 \\ d_{lev}(i,r)/max(l_i,l_r)) & \text{otherwise} \end{cases} \tag{2.4}$$

where:

(i) i is the name given as input to the algorithm.
(ii) r is the matched name in the blacklist.
(iii) l_r is the length of r.
(iv) l_i is the length of i.
(v) d_{lev} is the edit distance between i and r.

The formula can be read, as: the *similar scoring* $S_m(i)$ of the given name is the maximum score agains all the results r found in the blacklist.

2.6.2. *Terms expansion scoring S_t.* The terms expansion scoring will determine whether the terms of the name are included within a result of the blacklist or viceversa, i.e., it will determine if 'JUAN GARCIA' is included in 'JUAN LOPEZ GARCIA' and will be calculated as:

$$S_t = \max(M_{terms}/\max(i_{terms}, T_n terms)) \tag{2.5}$$

where:

(i) M_{terms} is the matched terms against a blacklist result n.
(ii) i_{terms} is the number of terms in the name given as input.
(iii) $T_n terms$ is the number of terms of an n blacklist name.

2.6.3. *Total scoring T_s.* The total scoring assigned to a matching operation will be calculated as the maximum scoring between S_m the *similar scoring* and S_t, the *terms scoring*:

$$T_s = \max(S_t, S_m) \tag{2.6}$$

3. Algorithm Implementation

Figure 1 shows a proposal for a high level implementation of the algorithm.

4. Features

Besides the main purpose of the platform that is provided a matching scoring against multiple blacklist in a heterogeneous way, there are some additional features exposed in the next sections that the platform will include to provide our customers with the tools and functionalities required to manage blacklist monitoring and screening of customers/entities.

(i) Blacklists management: configure how many blacklists available will be used by a determined customer.
(ii) Algorithm tunning: configure the different options to fine-tunning the algorithm's options to match the customer needs.
(iii) Whitelists: the customer should be able to define its own whitelists, as exceptions to the blacklists based on its own needs (person identifiers, person's names, identification numbers, etc.).
(iv) Monitoring and Analytics: tools required to analyse and monitor the operations.

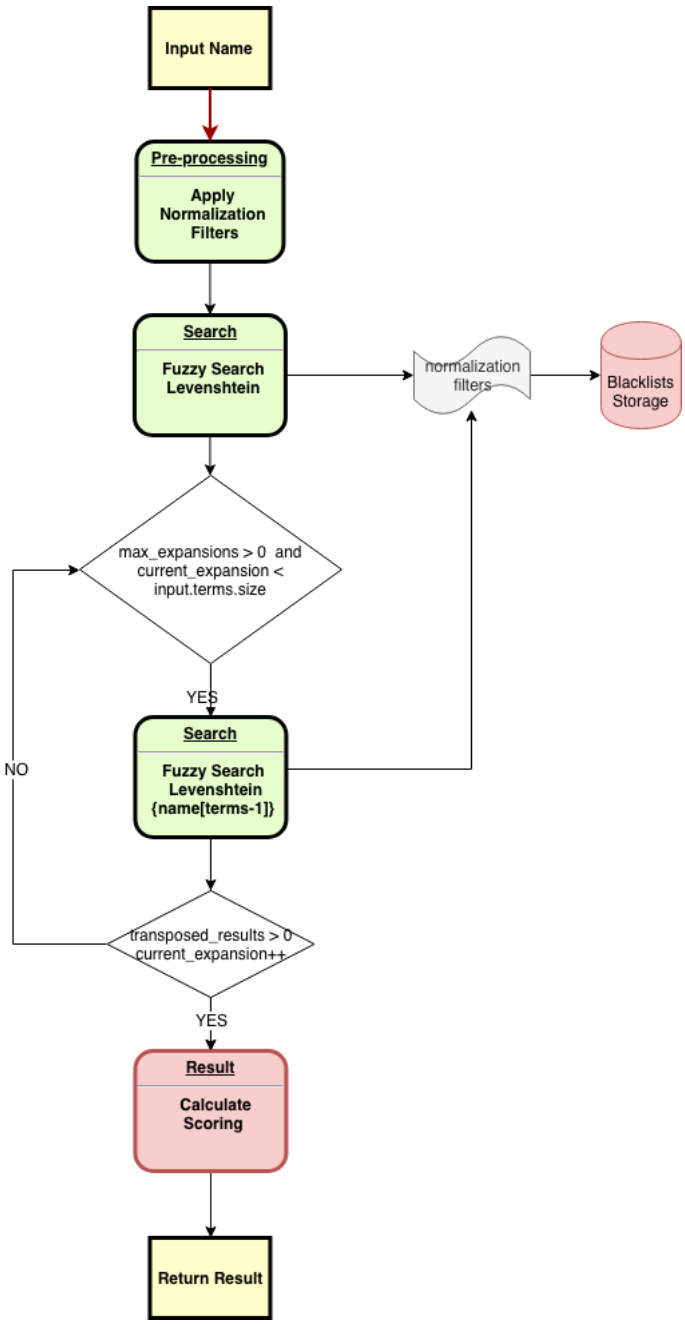

FIGURA 1. Flowchart algorithm

5. TECHNICAL ARCHITECTURE

When thinking in terms of the architecture to build and deploy the platform we have to think in advance what are the requirements and advantages of choosing one or another approach. In order to provide our financial customers with a solid and robust platform and based on our experience, we have seen in the past several challenges that need to be solved at technical level:

(i) **Setup Costs**. All software developments require a combination of programming language, design and system administration abilities. Usually any application/platform requires an integrated environment, someone will have to buy, build, design, manage and maintain the components for the application. Neither the work nor the components not provide for any distinguishing features of your software—they are simply required.

(ii) **Maintain control over the architecture**. Even if the architecture is provided by a cloud vendor, we could still retain control over configuration and management. This is a big advantage when developing and deploying our own application.

(iii) **Speed of application deployment**. A custom infrastructure usually involves the setup of new hardware and auxiliar configuration (networks, storage, security, etc.) to deploy the application. The time to be fully operational is increased in these cases.
(iv) **Updates**. Having the platform and the application deployed in many different customers as an isolated platform, usually involves extra resources to keep all the deployed instances up to date with security patches, hotfixes and new features.
(v) **Scalability**. The scalability of the platform should be estimated in advance, which usually incurs in additional unrequired costs.

5.1. **Architecture Proposal.** To solve all the previous issues, the proposal architecture is a cloud-based PaaS based architecture, based on micro-containers and/or serverless functions that will allow us to:

(i) **Business Centric**. Our customers should not worry about infrastructure, environments and or technical issues, this will be a fully-managed service.
(ii) **Multi tenancy**. All the data of every single customers should be isolated from any other customer, while providing a same interface to all of them.
(iii) **Cost optimized**. Using some of the current cloud providers will allow us to use the resources required for the actual volume, with almost unlimited growth.
(iv) **Fast deployments**: Having control of the entire infrastructure, will allow us to use performant, well tested DevOps pipelines, to provide frequent updates and fast solutions to the requirements of our customers.

5.2. **Microservice/serverless principles.** In the modern era, software is commonly delivered as a service: called web apps, or software-as-a-service. These principles (based on well know *12-factor app methodology*[1] are defined to build software-as-a-service apps that:

- Use declarative formats for setup automation, to minimize time and cost for new developers joining the project
- Have a clean contract with the underlying operating system, offering maximum portability between execution environments.
- Are suitable for deployment on modern cloud platforms, obviating the need for servers and systems administration.
- Minimize divergence between development and production, enabling continuous deployment for maximum agility.
- And can scale up without significant changes to tooling, architecture, or development practices.

[1]https://12factor.net

Just by looking at the goals, we can get a sense that having this would make development easier and more enjoyable. This is the way we should look at these factors. Not as a recipe how to design the full system, but rather a set of prerequisites that can get the project off to a great start:

(i) Codebase – *One codebase tracked in revision control, many deploys.* This is pretty fundamental, we should not create two different repositories when all you need to do is different setup for production.
(ii) Dependencies – *Dependencies.* Thinking about dependencies usually does not go beyond the dependant libraries. As long as you use a standard build tool (npm, yarn, maven, gradle, NuGet) you have the basics covered.
(iii) Config – *Store config in the environment.* One of the common mistakes is to distribute the configuration among the different modules. While this seems a good idea at the beginning, as the number of services grow, this can go quickly out of hand.
(iv) Backing services – *Treat backing services as attached resources.* Services should be easily changeable, i.e., having strong contracts with well isolated domains between services, will allow us to easily rewrite them in another technology, or replace them by newer versions easily
(v) Build, release, run – *Strictly separate build and run stages.*
(vi) Processes – *Execute the app as one or more stateless processes.* The idea of stateless services, that can be easily scaled by deploying more of them.
(vii) Port binding – *Export services via port binding.* This is quite simple: make sure that your service is visible to others via port binding.
(viii) Concurrency – *Scale out via the process model.* The idea is, as you need to scale, you should be deploying more copies of your application (processes) rather than trying to make your application larger.
(ix) Disposability – *Maximize robustness with fast startup and graceful shutdown.* Fast startup is based on our ideas about scalability and the fact that we decided on microservices. It is important that they can go up and down quickly. Without this, automatic scaling and ease of deployment, developments are being diminished.
(x) Dev/prod parity – *Keep development, staging, and production as similar as possible.*
(xi) Logs – *Treat logs as event streams.* Trends, alerts, heuristic, monitoring: all of these can come from well design logs, treated as event streams and captured by some well know technologies (Logstash, Spluk, etc..). Having a well monitored platform is essential.
(xii) Admin processes – *Run admin/management tasks as one-off processes.* Ship admin code with application code to provide admin capabilities. The tools

should be there even if they are not part of the standard execution of the service.

6. Technologies

6.1. **Development Frameworks.**

(i) *Spring boot 2* as base framework for microservices/serverless functions development.
(ii) *Spring Cloud Config.* Configuration management.
(iii) *Maven 3.* Dependency Management.
(iv) *JPA.* Persistence.
(v) *Open Feign.* HTTP Clients.
(vi) *Spring Cloud Sleuth.* Distributed tracing solution.
(vii) *Angular 6.* Web development.
(viii) *ElasticSearch.* Data storage for indexed blacklists.
(ix) *MongoDB/DynamoDB.* NoSQL Storage.
(x) *PostgreSQL.* E/R transactional storage.
(xi) *RabbitMQ.* Messaging Broker.

6.2. **DevOps Stack.**

(i) *Git* as DVCS.
(ii) *Jenkins* as automation server.
(iii) *Docker* as container's technology.
(iv) *Hashicorp Terraform* Infrastructure/Deployment.

6.3. **Infrastructure Stack.** Based on AWS Services.

(i) *AWS ECS.* Container orchestrator.
(ii) *AWS Lambda.* Serverless execution runtime.
(iii) *AWS API Gateway.* API management.
(iv) *AWS ElasticSearch.*
(v) *EC2 autoscaling.*

7. Phases of the project

Índice