Catchem: A Browser Plugin for the Panama Papers using Approximate String Matching

Panos Kostakos, Miika Moilanen, Arttu Niemelä, Mourad Oussalah
Center for Ubiquitous Computing
University of Oulu
Oulu, Finland
Email: panos.kostakos@oulu.fi, miika.moilanen@student.oulu.fi, arttu.niemela@oulu.fi, mourad.oussalah@oulu.fi

Abstract— **The Panama Papers is a collection of 11.5 million leaked records that contain information for more than 214,488 offshore entities. This collection is growing rapidly as more leaked records become available online. In this paper, we present a work in progress on a web browser plugin that detects company names from the Panama Papers and alerts the user by means of unobtrusive visual cues. We matched a random sample of company names from the Public Works and Government Services Canada registry against the Panama Papers using three different string matching techniques. Monge-Elkan is found to provide the best matching results but at increased computational cost. Levenshtein-based approach is found to provide the best tradeoff between matching and computational cost, while Jacquard index like approach is found to be less sensitive to slight textual change.**

Index terms— **Corruption; string matching; Panama Papers**

I. Introduction

Offshore tax heavens enable corruption and economic crime, and have a tremendous negative impact on European economy [1]. Law enforcement officers and journalists have various toolkits at their disposal for investigating economic crimes [2,3]. However, existing tools and methods are limited by: (1) offline processing and firewalls, (2) low customization, (3) poor search engine integration, (4) high technical expertise required, and (5) digital amnesia. As a result, end-users who make extensive use of open-source intelligence through online search engines cannot effectively leverage tacit knowledge accumulated from past investigations [2]. To address this lack of tools, we have developed *Catchem*, an online approximate string matching browser plugin that can unobtrusive flag out *entities of interest* while browsing a webpage. We run a computationally expensive brute-force method to compare company names from the Public Works and Government Services Canada (PWGSC) [4] against the Panama Papers [5] and successfully detected numerous exact matches. Especially, we compared three different approximate string matching algorithms to see which one would be the best for implementing a string matching plugin in Google chrome. The plugin would be used for detecting names, addresses, and company names that appear in the Panama Papers. The solution we are putting forward has three technical specifications: (1) string matching has to be tolerant to spelling mistakes or other fraudulent changes in the names, (2) the matching has to be fast to run swiftly on a browser, and (3) data customisation.

II. String Matching: Concepts & Applications

The problem of the approximate string-matching can be framed as follows. Given the query string (long pattern) $P = p_1p_2\ldots p_m$ and the short text string $T = t_1t_2\ldots t_n$ in alphabet Σ of size σ, find a substring $T(i..j) = t_i\ldots t_j$ which has the smallest Edit distance to the query string P [6]. The approximate occurrence of substring P in T is defined as the k edits (insertions, deletions, substitutions) in order to convert p_1 to t_1. Approximate string matching has been used in numerous criminological applications against identity fraud, phishing, economic fraud, and plagiarism [7]. Here we focus on a web-browser integration that could improve investigators' ability to use tacit knowledge in their work.

III. Context of the Study

The Panama Papers database is a collection of 11.5 million leaked documents that hold information about financial and attorney-client matters attained from Mossack Fonseca. Reporters have found out that some of the Mossack Fonseca shell corporations were used for illegal purposes, such as fraud, tax evasion, and avoiding international sanctions. For these reasons companies and people associated with the "Panama Papers" are more likely to be associated with criminal actions and corruption than average [5].

Catchem would be useful for journalists, law enforcement officers, government officials, and other parties that are likely to go through large amounts of open-source intelligence aiming to find possible criminal connections. For example, a journalist investigating corruption within government circles, would find it easing to not go through all the documents by hand. That said, the findings of this application should not be regarded as hard evidence. Instead this application should be thought as a guide, to help investigators commit resources in right direction. Moreover, a browser plugin was chosen as the means of implementing this application for a couple of reasons. First, the web is a vast source of information and harnessing that information brings great benefits. Second, installing and using browser plugins is very easy.

Catchem is a click-to-play plugin that could be activated by clicking a button on the top right of the browser, right next to the address bar. By highlighting correct matches, the plugin makes it easy to quickly browse through HTML documents to see if names or addresses in the documents are connected to the Panama Papers. However, building the application for browsers imposes some limitations and disadvantages. While the browser becomes faster by the day, it is not the most powerful computational resource available. As a result, our approach limits the use of vast amounts of memory. This means that the plugin should be lean and fast, in terms of computer clock cycles and memory consumption. Evidently, the approximate string matching is a key bottleneck to overcome.

The remaining of the paper provides results for the following three algorithms: (1) Levenshtein distance algorithm, (2) Monge-Elkan, and (3) Jaccard distance algorithm, also known as the Bag distance algorithm. We compared these three algorithms using three different measures: (1) time consumption, (2) memory consumption, and (3) quality of matches. These measures will be given a detailed description in the following sections.

IV. Method

A. Implementation and Metrics

The programming language of choice used to implement the software for testing the three algorithms was Python 3.6.1. Python offers a wide variety of libraries for natural language processing and data visualization, and has a comprehensive library for natural language processing called NLTK. Also, Python is a free and open source language, so it adds no financial load and allows for improvement of the language if needed.

In addition to the large variety of natural language processing tools, NLTK has a great documentation and free online resources for learning the conventional uses of the tools it provides. We used *NTLK.metrics* for measurement metrics and *NTLK.word_tokenize* for word tokenization. Another important tool we used was the Python library *py_stringmatching*. This Python package provided us with the three approximate string matching algorithms Levenshtein, Monge-Elkan, and Jaccard. For memory consumption measurements, we used a Python package named *memory_profiler*. It can be used to measure the memory consumption of single Python functions or whole Python programs. This package was the only memory profiling tool we could find that offered us the functionality we needed. *Time* module, that ships with Python, was deployed for time measurements.

We used a test dataset with over 35000 names to check against the Panama Papers. We measured the time it took to match one name from the dataset against all the names in the Panama Papers by calling the *time* method of the time module before and after the matching and saving those times into variables *t0* and *t1*, where t0 would be the time in seconds before matching and time in seconds after matching. The difference between t1 and t0 is the time spent on finding a match for a single name. We used the same method for measuring the time it took to go through a whole list of names of a given size.

Memory profiling was done with the *memory_profiler* package. It is used by attaching a decorator to a function and the package would then profile the memory usage automatically and print a report in a file. Respectively, to measure the quality of the matches we categorized them in four groups: (1) over 99% matches, (2) over 90% matches, (3) over 70% matches, (4) over 50% matches, and (5) less or equal to 50% matches. Our approach to cluster the results into these categories provides additional value to the overall functionality of our plugin as it enables investigators to generate and prioritize leads. Thus, in the first category we expect to find true positive matches or "smoking gun" cases, whereas in the lower bands we expect to capture company names that are either misspelled or tempered to avoid detection.

B. Data Sources

Two sets of data were chosen for testing the string matching algorithms. The first data set is the openly available Panama Papers that provides information on companies, entities and individuals, of which some are implicated in illegal financial activities [5]. The other data set is the Public Works and Government Services Canada, a database maintained by the Canadian government, holding identifying information on companies and entities operating and having contracts in Canada, and which are also openly available [6].

The Panama Papers are available as a collection of .csv files (comma separated values). The collection includes single files for legal entities, intermediaries, officers, and addresses. The legal entities were used as they hold mostly names of companies and organizations. Overall, the entities file holds 495,039 lines of entities, although some of the entries are about the same company name. All data is in string format, the entity names being in capital letters.

The Public Works and Government Services Canada database is also available via an official web portal. The data that was used in this paper is contract data, which holds information on contracts awarded to companies and other suppliers by Public Works and Government Services Canada, that were over 10,000$ in value. The data is available since 2009 for every fiscal year. The database can be searched through its web portal, but raw data can also be downloaded as .csv files. For this project, .csv file for fiscal year 2016-2017 was used. Each line provides a 'supplier standardized name', the name of the company that the contract was awarded for, which were used as the list of names to match. Data was in string format, with the supplier names being capitalized..

C. Data Pre-Processing

Preprocessing was conducted on the data in a few ways. First, the company or entity names were extracted from each data set to reduce the amount of unnecessary data. Then, all the names were edited to be completely lowercased in order to standardize the data. Finally, many companies share common identifiers like 'corporation', or 'technologies', which would result in a large number of false positives, especially if the common part is much longer than the actual name of the company. Thus, common terms were removed. The removed words were chosen so that they were common (included in more than 1% of the names) in both of the data sets, or very common (included in more than 10% of the names) in one of the data sets. By this reasoning, the following 25 stop-words were deleted from the names: associates, canada, company, consultants, consulting, corp, corporation, development, enterprises, group, holding, holdings, inc, incorporated, international, investment, investments, limited, ltd, management, services, systems, technologies, technology, trading.

After cleaning the data, a total of 481,720 names were left in the Panama Papers and 38,864 names in the Canada data set. We compared the frequencies of the original data with the cleaned data to verify that the removal of commonly used words from the lists did not change the overall quality of the data. Final testing data set was a total of 400 random names, with 200 from each of the two lists. Next, 200 of the names were randomly selected from the Panama Papers list, forming the list of 'bad' companies. Another 180 names were randomly selected from the 'Canada' list, forming the list of 'good' companies. The remaining 20 names in the 'Canada list' were extracted from the Panama Papers list, so that there would be 100% matches, forming the true positives for evaluation.

D. Matching Algorithms

We compared three similarity measurement algorithms that associate the likeness between two strings using different methods. First, we used the Jaccard distance algorithm, also known as the Bag distance algorithm. This algorithm obtains the non-common characters of two strings. It does so by dividing the characters in strings s1 and s2, and ordering them in two charsets X and Y, and corresponding differences X-Y and Y-X. Then the largest difference between s1 and s2 is computed by:

$$bagdistance(s1, s2) = max(|x\text{-}y|, |y\text{-}x|) \quad (1)$$

The similarity function of Jaccard distance is given by the following formula [8]:

$$simbag(s1, s2) = 1 - \frac{bagdistance(S1,S2)}{max(|S1|,|S2|)} \quad (2)$$

The second algorithm, Monge-Elkan [9] is a simple method for measuring similarity between two strings. The formula is as follows:

$$MongeElkan(A, B) = \frac{1}{|A|}\sum_{i=1}^{\|A|} max\{sim'(A_i, B_j)\}_{i=1}^{|B|} \quad (3)$$

Monge-Elkan needs a similarity function to measure the similarity, as can be seen from the formula above. The similarity function we used in this project was *Jaro-Winkler*.

The last algorithm we included in our comparison was the Levenshtein algorithm. It allows insertions, deletions and replacements in the measured strings [10]. This is advantageous since the plugin we are developing cannot be fooled by simple manipulations of the strings. Levenshtein distance formula is as follows:

$$levensthein(s1, s2) = 1 - \frac{levenstheindist(S1,S2)}{max(|S1||S2|)} \quad (4)$$

V. Performance Evaluation

The analysis was implemented using a custom Python software. The software takes any given list of strings as input and compares them using the chosen algorithms. The algorithm is chosen manually from the three included string matching algorithms; Monge-Elkan, Jaccard, and Levenshtein. Also, desired number of samples from the lists can be defined. The software measured time and memory consumption of the algorithms in order to compare them. Next, the robustness of the algorithms was compared by having them match the lists of modified names to the list of original names. The software runs the chosen algorithm on the data and saves the match percentages and execution times for each individual pair of words. Additionally, the memory consumption of the string matching function is measured and saved by memory profiling Python library called *memory_profiler*.

Fig. 1a shows the total execution time of matching all the 200 names in the Panama-list with the 200 names in the Canada-list. This totaled in 40,000 pair of names to match. Jaccard and Levenshtein are similarly fast, executing in under 5 seconds. However, Monge-Elkan is considerably slower, being more complex algorithm, and taking almost 60 seconds to execute. Fig. 1b illustrates the memory consumption of each algorithm. There are no significant differences in the memory usage. All the algorithms consume roughly 100 megabytes during execution.

Fig. 2 presents each algorithm's execution time for each individual word matched to all the 200 words in the other list. Results show that behavior of Levenshtein and Jaccard are similar, as in the total execution times. Both oscillate between to similar values. However, Monge-Elkan shows very different behavior. Because it is more complex algorithm, the execution time varies dramatically between each of the words.

Fig. 3 shows the results of testing the robustness of the algorithms. Robustness was defined as the ability to detect a matching pair of names, even if the other name is modified. Modifying was done as either subtracting letters from the beginning or the end of the name, or as padding the name with 'x'-letters. In the figure, 'orig' in the middle shows the results for original names, without modifications. Bars on the left-wing show results for subtracting 1 to 4 letters and bars on the right-wing show results for padding 1 to 4 letters. For example, company name 'pepsi' after subtracting 1 letter would be 'peps', and after 2 letters 'eps'. Padding with 1 letter would make it 'pepsix' and padding two letters 'xpepsix'. As shown in Fig. 3, Monge-Elkan is the least sensitive to modified names with average similarity being over 70% even with 4 subtracted letters. Levenshtein is also quite robust, with the matching result falling a little faster than Monge-Elkan's. Jaccard was very sensitive to name modifications. Its matching result crashed down to under 20% with only one letter subtracted or padded**.**

Table 1 presents the complete raw matching results for the three algorithms. Results are categorized by which percentile their match result belonged.

TABLE I. RAW MATCHING RESULTS

Alg.	Total (sec)	Mem.	>99%	>90%	>70%	>50%	=<50%
Me	18.99339628	109MB	20	2	870	19170	19938
Lev	11.30473184	105MB	20	0	4	109	39867
Jac	9.571173906	107MB	20	0	0	1	39979

Columns contain the following information: (Alg) Name of the algorithm; (Total) Total number of seconds it takes to complete the string matching for each algorithm; (Mem.) Total memory used by the function including loading of raw data, modules, and function decorator; (>99%) Number of matches between the two lists with a ratio greater than 99%. Given that we have manually injected 20 perfect matches, this column represents true positive matches. In real-life situation these are "smoking gun" cases; (>90%) Number of matches between the two lists with a ratio greater than 90%. These cases are potential false positives, and in real-life situations these are considered to be leads that investigators might want to follow-up; (>70%) Number of matches between the two lists with a ratio greater than 70%; (>50%) Number of matches between the two lists with a ratio greater than 50%. (=<50%) Number of matches between the two lists with a ratio less or equal to 50%. Matches below the 90% ratio threshold provide very few actual leads.

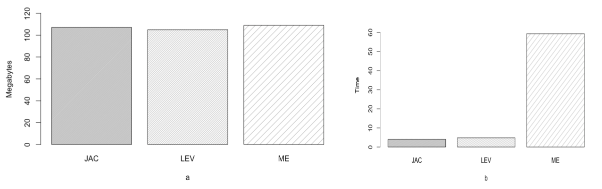

Fig. 1. 1a) Memory usage of each algorithm. 1b) Total execution times of each algorithm.

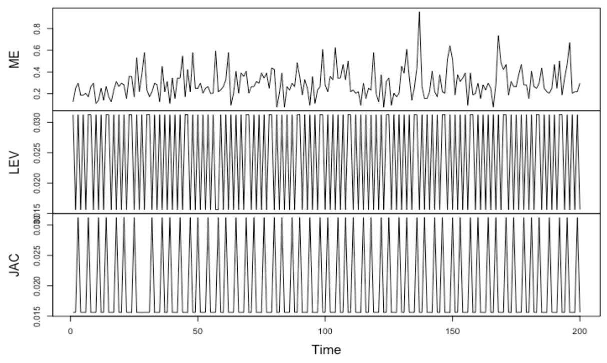

Fig. 2. Individual string matching times Monge-Elkan (ME), Levenshtein (LEV), and Jaccard (JAC).

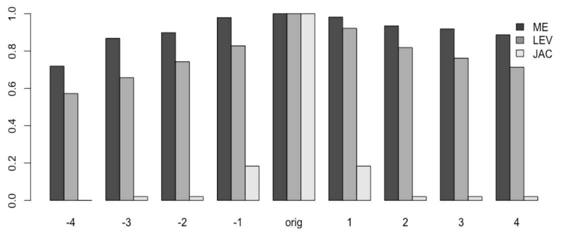

Fig. 3. Effects of modifications. Positive numbers indicate addition of letters; negative numbers indicate subtraction of letters; and "orig" indicates two identical strings.

VI. CONCLUSION & RESULTS

The long-term objective is to build a plugin that could highlight persons and entities of interest while browsing open source intelligence on the internet. In this paper, we tackle the largest bottleneck of our objective, namely understanding how different approximate string matching algorithm respond to a specific lexical domain.

By using the methods, tools and data described in detail above we were able to come in to a conclusion that, for the purposes of the proposed plugin, the best algorithm to use would be the Levenshtein distance algorithm. Levenshtein distance was not the fastest algorithm, nor did it give us the best matches, but the tradeoffs it imposes were in the best balance. Compared to Monge-Elkan the matching was not quite as good, but Levenshtein did produce less false positives. Levenshtein was not as fast as Jaccard distance but it came very close. In terms of memory consumption all the algorithms were neck to neck. Our proposed approach has a number of limitations that future work might seek to address.

First, to improve the preprocessing of company names more robust tools are required that will focus on extracting lexical features from the specific domain while controlling for the overall quality of the data. Second, the scope covered in the paper can expand to include more algorithms along with custom-made programs that deal with the memory consumption more systematically. Third, there is a large number of company databases that can be used to improve the detection capabilities of the algorithms.

Finally, further work would include conducting user studies that measure the effectiveness of our tool, and making the plugin even faster by adopting big data processing techniques, such as hash maps and multiple core processing. The plugin would also benefit if the data could be inserted in to a distributed database as this would allow custom datasets to be utilized by law enforcement and journalists.

REFERENCES

[1] European Commission, EU Anti-Corruption Report, Brussels, 3.2.2014, COM(2014) 38 final, 2014.

[2] P. Gottschalk, Knowledge Management Systems in Law Enforcement: Technologies and Techniques. London: Idea Group Publishing, 2007.

[3] U.S. Department of Justice, Investigative Uses of Technology: Devices, Tools, and Techniques. Washington: U.S. Department of Justice Office of Justice Programs, October 2007.

[4] Government of Canada, Public Works and Government Services Canada (PWGSC), http://open.canada.ca/data/en/dataset/53753f06-8b28-42d7-89f7-04cd014323b0

[5] Offshore Leaks Database, https://offshoreleaks.icij.org/

[6] G. Navarro, "Approximate string matching," Encyclopaedia of Algorithms, pp: 1-5, 2014.

[7] G. Wang, H. Chen, H. Atabakhsh, "Automatically detecting deceptive criminal identities," Communications of the ACM, vol 47, 2004.

[8] P. Angeles and L.F. Perez-Franco, "Analysis of String Comparison Methods During De-Duplication Process," ICIEV Proceedings, 2015

[9] A. Monge and C. Elkan, "An efficient domain-independent algorithm for detecting approximately duplicate database records," SIGMOD workshop, 1997.

[10] G. Navarro, "A guided tour to approximate string matching," ACM computing surveys (CSUR), vol: 33, 2001.