

A Project Report
on
Matches: Matching Data

Submitted by
Saumay Rustagi (Registration No. 219301636)
III Sem. – B.Tech. - CSE

in partial fulfilment for the award of the degree
of

BACHELOR OF TECHNOLOGY

In
Computer Science and Engineering



**MANIPAL UNIVERSITY
JAIPUR**

**Department of Computer Science and Engineering,
School of CSE,
Manipal University Jaipur,
Nov 2022**

Objective

A common problem in data analysis is the same type of data from different sources is categorized differently, which causes discrepancies and inconsistencies. Correctly recognizing when the same type of data is being labeled differently – specifically employee data on payslips - and automating this process, is the objective of the project. This is explained further below.

For example, the two labels below measure the same quantity but would be treated as completely different by an OCR.

Sadhana Infotech
#335, 19th Main, Rajaji Nagar, 1st Block, Bangalore-560010
Wage Slip for the month of Feb/2019

Emp ID : 16
PF. No. : KN/45889/1016
Department : Manufacturing
PAN : JOPPL89895

Employee Name: SANDHYA
Designation : Operator
A/c No : sb-1041
Mode of Pay : State Bank of India

Earnings	Rate	Amount	Deductions	Amount
BASIC	20,000.00	20,000.00	PF	2,880.00
DA	4,000.00	4,000.00	PT	200.00
HRA	9,600.00	9,600.00	TDS	4,042.00
CONV	800.00	800.00		
SPL ALLOW	5,600.00	5,600.00		
Total	40,000.00	40,000.00	Total	7,122.00

Salary Slip NOV - 19

Emp No : CSE-8182
Name :
PF No : N/A
Bank Acc No :
UAN Number :
Location : ANDHERI
WORKDAYS : 30

DOB : 02/01
DOJ : 11 May 2017
PAN NO :
Bank : HDFC BANK
IFSC Code : HDFC0001024
Designation : C.S.E.
DaysInMonth : 30

Earnings	Rs.	Deduction	Rs.
Basic	4,000.00	Professional Tax	200.00
Conveyance	5,000.00	Employee PF	680.00
Performance Allowance	5,000.00	Income Tax	-
Fuel Allowance	-		
Re-employment	-		
Total Earnings	25,000.00	Total Deduction	880.00
Net Pay : 23,620.00/-			
In Words : Twenty Three Thousand Six Hundred Twenty Rupee Only.			

'Emp ID' and 'Emp No' are different labels for the same data.

The purpose of my project is to help streamline data, specifically employee data on payslips. The same employee information (ID numbers, financial information, etc) is often stored under different names. Companies use different labels for employee data on their salary pay slips. OCR treats these labels as distinct objects even when they refer to the same information. This leads to inconsistent processing of data.

These inconsistencies multiply and pose problems when trying to parse data on a large scale. My goal is to reduce these inconsistencies by recognizing that these data points come under the same category and correctly sorting and labelling them.

Methodology

The project's approach is to maintain a set of standard labels for the data and try to correctly assign these labels to new data that is received.

We compare the 2 words against each other assuming that the first word is treated as "correct".

Here, we specify a certain Levenshtein distance (*the minimum number of single-character edits (insertions, deletions or substitutions) required to change one word into the other*) between two words to ascertain how similar they are. The lower the distance, the higher their similarity.

We take the word with the least Levenshtein distance (under a certain upper bound) from the input and store its value into the corresponding field.

We split the 2 words into 2 distinct arrays of type string and compare each string against the other to ascertain the strings to match and then compute their Levenshtein distance.

Not only does it allow payslip terms to be corrected, it even accounts for spelling errors, though only to the extent where the word doesn't change its meaning.

Flowchart



Start

psvm

Str a = args[0]
Str b = args[1]

a = a.strip()
b = b.strip()

float c = new Levenshtein().get_lew(a, b); *

S.O.P $C \geq 1 + C$

Stop

class
Matches

Lexington

* net-lex(stra, strb)

$$\begin{aligned} \text{CS1} &= a \cdot \text{pld}(\text{---}), \\ \text{CS2} &= b \cdot \text{pld}(\text{---}), \end{aligned}$$

↑ obj. konstante $\sin(\alpha_1, \alpha_2)$

Note, 'String' has been shortened in some places for brevity.

- Interface needed inside Verneblein

Forest translation - diet (String a, String b):

Doat. `while - true {String() arr1, String() arr2},`

- ↳ implemented by Computation (class) nested inside Levenstein

* | launchen - ein (Strang[Var1], Ste[Var2])

```

    boolean flag = false;

```

float dimen = 0

* \downarrow levenshtein - dist (String a, String b)

int n1 = a.length();
int n2 = b.length();

float dist = Math.max(n1, n2) - Math.min(n1, n2);

return

Inbuilt Math functions
with custom implementation.

n1 <= n2

true

n2 returned

false

n1 returned

i = 0

i < Math.min(n1, n2)

a.charAt(i) != b.charAt(i)

default

true

+dist

try block

dist = dist / (n1 + n2)

catch ArithmeticException

S.O.P. "Something went Wrong during Calculation!"

return 1 - dist

Inbuilt function
that safely traverses
a String until the
given index & returns
the character at the
index.

n2 returned
true
n1 > n2
false
n1 returned
return

Code

Main file: Matches.java

```
import java.util.Scanner;

import java.lang.Math;


class Matches {

    static Scanner in = new Scanner(System.in);

    public static void main(String[] args) {

        int tt = 4;

        while (tt-- > 0) {

            solve();

        }

    }


    static void solve(){

        String a = in.nextLine();

        String b = in.nextLine();


        // Strip Whitespace

        a = a.strip().toLowerCase();

        b = b.strip().toLowerCase();


        float c = new Levenshtein().ret_lev(a, b);

        System.out.println((c >= 1) + " " + c);
```

```
}  
  
}
```

```
class Levenshtein {
```

```
float ret_lev(String a, String b) {
```

```
String[] s1 = a.split(" ");
```

```
String[] s2 = b.split(" ");
```

```
InnerStandardize obj = new Computation();
```

```
return obj.lvnshtn_sim(s1, s2);
```

```
}
```

```
interface InnerStandardize {
```

```
float lvnshtn_dist(String a, String b);
```

```
float lvnshtn_sim(String[] arr1, String[] arr2);
```

```
}
```

```
class Computation implements InnerStandardize {
```

```
public float lvnshtn_dist(String a, String b) {
```

```
/*
```


* Takes 2 strings and returns their levenshtein distance

*/

```
int n1 = a.length(), n2 = b.length();
```

```
float dist = Math.max(n1, n2) - Math.min(n1, n2);
```

```
for (int i = 0; i < Math.min(n1, n2); i++)
```

```
if (a.charAt(i) != (b.charAt(i)))
```

```
++dist;
```

```
try {
```

```
dist = dist / (n1 + n2);
```

```
} catch (ArithmeticException e) {
```

```
System.out.println("Something went Wrong during Calculation!");
```

```
}
```

```
return 1 - dist;
```

```
}
```

```
public float lvnshtn_sim(String[] arr1, String[] arr2) {
```

```
/*
```

```
* arr1 = an array of Strings from correct
```

```
* arr2 = an array of Strings from input
```

```
*/
```

```
boolean flag = false;

float lev = 0;

float dissim = 0;


for (int i = 0; i < arr1.length; i++) {

    flag = false;

    for (int j = 0; j < arr2.length; j++) {

        if (arr1[i].charAt(0) == arr2[j].charAt(0)) {

            float sim = lvnshtn_dist(arr1[i], arr2[j]);

            if (sim == 1.0)

                flag = false;

            lev += sim;

            break;

        } else

            flag = true;

    }

    if (flag == true)

        ++dissim;

}

try {

    dissim = dissim / Math.max(arr1.length, arr2.length);

} catch (ArithmeticException e) {

    System.out.println("Something went Wrong during Calculation!");

}

return lev - dissim;

}

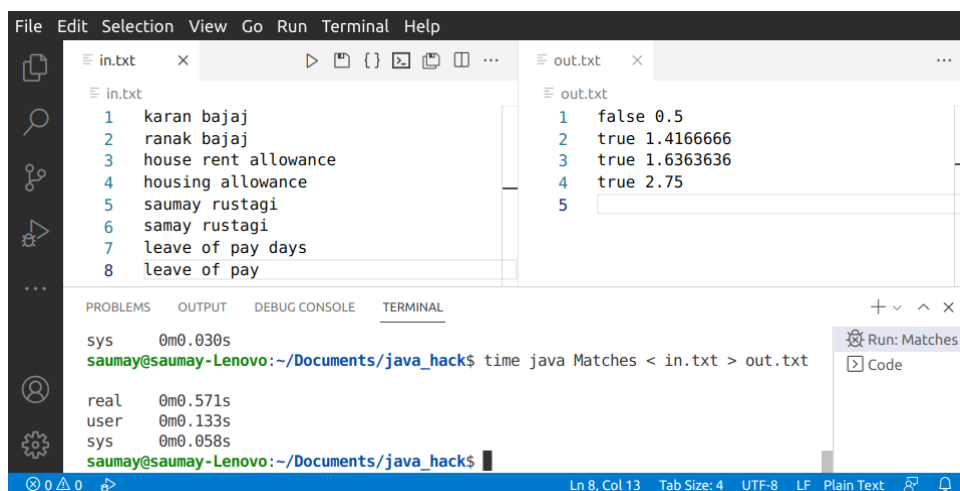
}
```


System Requirements

The program was run on a Java JDK implementation accessed by APT and built by the Ubuntu repository based on the openjdk 17.0.5 specification on a Lenovo ideapad S145 with an AMD Ryzen 5 3500U chipset running Ubuntu MATE 22.04.1 LTS x86_64 with a usable memory of 5806 MiB.

However, due to its lightweight, the program can be run on much slower hardware as long as it supports the openjdk 17.0.5 specification.

Output



The screenshot shows an IDE with two open files: `in.txt` and `out.txt`. The `in.txt` file contains 8 lines of input data, and the `out.txt` file contains 5 lines of output data. Below the files is a terminal window showing the command `time java Matches < in.txt > out.txt` and its execution time.

File	Line	Content
in.txt	1	karan bajaj
	2	ranak bajaj
	3	house rent allowance
	4	housing allowance
	5	saumay rustagi
	6	samay rustagi
	7	leave of pay days
	8	leave of pay
out.txt	1	false 0.5
	2	true 1.4166666
	3	true 1.6363636
	4	true 2.75
	5	

```
sys      0m0.030s
saumay@saumay-Lenovo:~/Documents/java_hack$ time java Matches < in.txt > out.txt

real    0m0.571s
user    0m0.133s
sys     0m0.058s
saumay@saumay-Lenovo:~/Documents/java_hack$
```

The program iterates through the 1st 2 lines 4 times and matches them against each other, We use the casting property of UNIX to pass in the input and output files that need to be parsed and updated respectively.

As we can see, not only does it discern that “ranak” and “karan” are different names, but it can also spot the spelling mistake in “saumay” and “saumay”.

The time utility lets us know that this is an extremely efficient program, clocking in at only 0.058s inside the actual runtime system; and that includes the input and output passing which could be optimized further for a trade-off with IO safety.

Conclusion

There are many reasons we would want to streamline data in this way to automate the process of payslip processing. Often data from payslips must be manually entered to be standardised, or if employment history needs to be organised it must be performed manually. Being able to automate payslip processing would also make employee verification faster.

This system can be used with a powerful OCR to automate the process of scanning and standardizing data more. Simple OCRs (Optical Character Recognition) fail at the task of accurately recognizing and sorting data labels, and a good OCR combined with strong text processing could overcome this problem.

Several improvements can be made to project. The combination of algorithms in the code do not account for every case. Sometimes the data label is too different for the algorithms to recognize, and such slip cases can be avoided by making the algorithms more robust.

Nevertheless, the project achieved its goal of discerning when two different data labels point to the same data, and when they don't.

References

Levenshtein Distance: https://en.wikipedia.org/wiki/Levenshtein_distance

Edit Distance: <https://www.geeksforgeeks.org/edit-distance-dp-5/>

Payslip Data Examples: <https://images.google.com/>