

Deduplikation

Daniel Stefan Heinz-Eugen Klose
TU Dortmund University
daniel-stefan.klose@udo.edu

ABSTRACT

Ein wichtiger Teil des ETL-Prozesses ist die Deduplikation. In dieser Ausarbeitung werde ich mich genauer mit dem Thema der Deduplikation befassen. Es werden mehrere Metriken vorgestellt, an denen man Duplikate erkennen kann. Derweiteren werde ich eine Methode vorstellen mit der man diese effizient implementieren kann.

PROBLEM

In einem OLTP-System sind die Daten meist "schmutzig". Das bedeutet, dass das ein Großteil der Daten auch nach wie vor mit der Hand von einzelnen Personen in das Datenbank-System eingetragen werden.

Bei diesen eingetragen passieren verschiedene Fehler. Ein Fehler davon ist, dass ein Eintrag falsch geschrieben ist bzw. mit verschiedenen Schreibweisen in dem Datenbank-System zu finden ist. Ein Beispiel ist dafür ist die Schreibweise für Dortmund. Ein Mitarbeiter eines Unternehmens macht unabsichtlich einen Schreibfehler und trägt Dortmund als Verkaufsort eines Produkts.

Dies führt dazu, dass nach der Überführung der Daten in ein Datawarehouse dieser Verkauf nicht zu den Verkäufen des Standortes Dortmund gehört und damit auch nicht bei den Aggregationen mit eingerechnet wird. Hier kommt die Deduplikation im ETL-Prozess ins Spiel.

LÖSUNG

Das Finden und Zusammenführen von solchen Duplikaten wird im ETL-Prozess auch Deduplikation genannt. Bei der einfachsten Form für das Finden von Duplikaten vergleichen wir, mit einem vorher festgelegtem Maß jedes Wort mit jedem anderen und führen diese zusammen welche unterhalb eines bestimmten Thresholds liegen.

Im weiteren Verlauf dieser Arbeit werden wir sehen, dass dies eine sehr ineffiziente Lösung ist und schauen uns weitere Alternativen an.

Vergleichsmaße

Als Erstes wollen wir uns verschiedene Vergleichsmaße ansehen, wie auch die verschiedenen Ansätze dieser.

Das wohl einfachste Maß für die Gleichheit zweier Zwischenketten ist die sogenannte *Editierdistanz* [2, Vgl. S. 2], welche auch als *Levenshtein-Distanz* bezeichnet wird. Die Editierdistanz misst wie viele Operationen es braucht um den einen String in den anderen zu transformieren. Die Operationen lauten *erstzen*, wobei ein Zeichen durch ein weiteres ersetzt wird, *einfügen*, wobei ein Zeichen in einen String eingefügt wird und *löschen*, wobei ein Zeichen gelöscht wird. Um dies zu implementieren machen wir Gebrauch von dem Konzept der *dynamischen Programmierung*. Der Algorithmus ist in Pseudocode 1 zu sehen [1, Vgl. S. 223].

Pseudocode 1 Editierdistanz mit dynamischer Programmierung

```
1: procedure DISTANCE( $u, v$ )
2:    $m := |u|$ 
3:    $n := |v|$ 
4:   decalre  $d[0..m, 0..n]$ 
5:   for  $i$  from 0 to  $m$  do
6:      $d[i, 0] := i$ 
7:   for  $i$  from 0 to  $n$  do
8:      $d[0, i] := i$ 
9:   for  $i$  from 1 to  $m$  do
10:    for  $j$  from 1 to  $n$  do
11:      if  $u[i] = v[j]$  then
12:         $cost := 0$ 
13:      else
14:         $cost := 1$ 
15:       $d[i, j] := \min($ 
         $d[i - 1, j] + 1,$ 
         $d[i, j - 1] + 1$ 
         $d[i - 1, j - 1] + cost)$ 
return  $d[m, n]$ 
```

Als weiteres Maß für das Finden von Duplikaten ist die *Jaccard Similarity* [4, Vgl. 4.1.1]. Diese misst die Ähnlichkeit von Mengen. Da wir aber keine Mengen vergleichen sondern Strings, machen wir einen kleinen Trick und überführen unsere Strings in Mengen, indem wir jeweils zwei aufeinanderfolgende Zeichen des Strings zu einem Element der Menge machen. Zum Beispiel wird aus "Sweet" dann {"Sw", "we", "ee", "et"}. Damit können wir auch Strings vergleichen. Das Verleichen selber geschieht mit der Formel:

$$\frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}$$

Als Beispiel vergleich wir *Sweet* und *Sweat*.

$$\frac{| \{ "Sw", "we", "ee", "et" \} \cap \{ "Sw", "we", "ea", "at" \} |}{| \{ "Sw", "we", "ee", "et" \} \cup \{ "Sw", "we", "ea", "at" \} |} = \frac{| \{ "Sw", "we" \} |}{| \{ "Sw", "we", "ee", "et", "ea", "at" \} |} = \frac{1}{3}$$

Neben diesen Maßen existieren weiter, welche sich an der Aussprache von Worten orientiert. Hierfür ist der Soundex [3, Vgl. S. 5] sowie Metaphone [3, Vgl. S. 5] ein Beispiel. Beim Soundex wird das erste Zeichen beibehalten und die folgenden Zeichen werden durch die jeweilige Nummer ersetzt. Andere Zeichen werden entfernt. Doppelt aufeinander folgende Zahlen werden auf eine Reduziert. Es werden maximal drei nummer verwendet. Falls weniger vorhanden sind werden sie mit Nullen aufgefüllt.

$$f(n) = \begin{cases} 1 & \text{falls } n \in \{b, f, p, v\} \\ 2 & \text{falls } n \in \{c, g, j, k, q, s, x, z\} \\ 3 & \text{falls } n \in \{d, t\} \\ 4 & \text{falls } n \in \{l\} \\ 5 & \text{falls } n \in \{m, n\} \\ 6 & \text{falls } n \in \{r\} \\ - & \text{sonst} \end{cases}$$

Vergleichsmatrix

Um jeden String mit jedem anderen zu vergleichen Stellen wir eine sogenannte Vergleichsmatrix auf. Haben wir eine List l an Strings haben wir eine Matrix d mit $|l| \times |l|$ Elementen. Da alle Vergleichsmaße kommutative sind und der Vergleich mit sich Selbst irrelevant ist, müssen wir jedoch nur die Einträge oberhalb der Diagonalen von $d[0, 0]$ bis $d[|l|, |l|]$ betrachten. Dies ist in Abbildung 1 dargestellt. Hierbei sind nur die Roten Zellen Interessant und müssen berechnet werden.

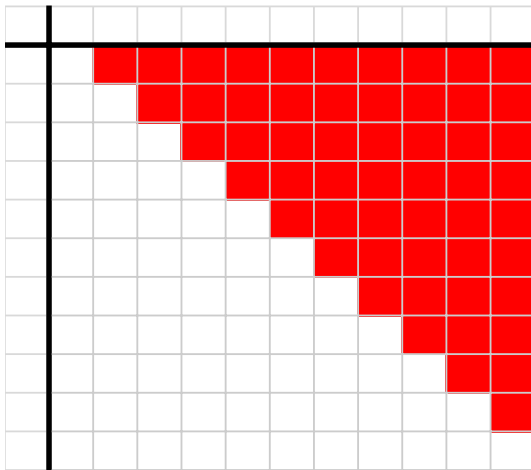


Figure 1: Volle Vergleichsmatrix

Dabei sieht man auch sofort das Problem der vollen Vergleichsmatrix. Es müssen sehr viele verglichen gemacht werden. Bei n Strings müssen

$$\frac{n^2}{2} - n$$

Vergleiche gemacht werden. Bei 1.000 verschiedener Elemente müssen 499.000 bei 10.000 schon 49.990.000 Vergleiche gemacht werden. Dies wird sehr schnell ineffiziente.

Die effizientere Lösung ist hier das Blocking [3, Vlg. S. 11]. Beim Blocking berechnen wir nicht die gesamte Vergleichsmatrix, sondern wir teilen die Liste der Strings in Blöcke auf. Die Vergleiche finden nun nur noch in diesen Blöcken statt, welches in Abbildung 2 zu sehen ist, oder es wird ein Fenster einer bestimmten Blockgröße über die Diagonale geschoben und verglichen werden nur innerhalb dieses Fensters gemacht.

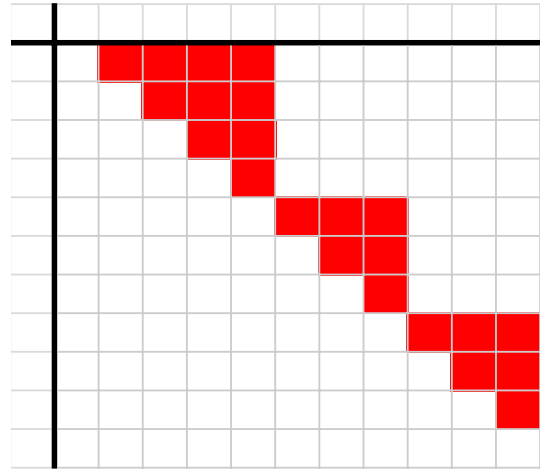


Figure 2: Vergleichsmatrix mit Blocking

Wenn wir jedoch nur die Liste strikt in n Teile aufteilen verlieren wir an Genauigkeit, da wir manche Strings nicht vergleichen. Um das zu verhindern teilen wir nicht einfach in n Blöcke, sondern wir sortieren die Strings anhand von bestimmten Kriterien. String einer bestimmten Kriterie kommen einen Block bzw. werden anhand dieser sortiert [3, Vlg. S. 11]. Diese Kriterien-Blöcke verhindern, das ähnliche, bzw. ähnliche String in einen Block kommen und verglichen werden. Es gibt mehrere Strategien Kategorien zu wählen. Diese kommt immer auch den Kontext des Strings an. Eine Strategie ist es nach dem Anfangszeichen zu sortieren, da dort meist kein Fehler gemacht wird. Eine bessere Strategie ist jedoch sich die Entity, die diesen String beschreibt, sich genauer anzusehen, um dort eine Eigenschaft zu finden, welche sehr Fehlertolerant und 2 String mit gleicher Bedeutung die selbe Eigenschaft haben. Nehmen wir zum Beispiel das deduplizieren von Straßennamen. Hier könnte eine sinnvolle Eigenschaft sein die ersten 3 Ziffern der Postleitzahl zu nehmen, da diese oft richtig Eingegben werden und zwei Straßennamen mit selber Bedeutung immer die selbe Postleitzahl hat. Wenn wir die n Strings in b gleich große Blöcke einteilen erhalten wir eine Laufzeit von

$$\frac{1}{2} \left(\frac{n^2}{b} - n \right)$$

Durch das aufteilen in Blöcken nach verschiedenen Eigenschaften reduzieren wir die Wahrscheinlichkeit, dass wir manche Duplikate nicht finden.

Ein weitere Erweiterung davon wäre wenn man diese Aufteilung in Blöcke und das finden von Duplikaten mehrfach mit verschiedenen Eigenschaften durchführt [3, Vlg. S. 11]. Dadurch sinkt die Wahrscheinlichkeit von Duplikaten im Endresultat nocheinmal.

ZUSAMMENFASSUNG

Alles zusammen ist die Deduplikation eine gute Idee. Jedoch muss eine Kontext bedingte Strategie entwickelt werden um Duplikate sicher und effizient finden zu können. Hierbei ist zuerst die Wahl

des Vergleichsmaß entscheidend. Des weiteren muss eine Fehler-tolerante und mit der Bedeutung übereinstimmende Eigenschaft gefunden werden, um die String in vernünftige Blöcke einteilen zu können.

REFERENZEN

- [1] P. Bille. A survey on tree edit distance and related problems. *Theoretical computer science*, 337(1-3):217–239, 2005.
- [2] W. W. Cohen, P. Ravikumar, S. E. Fienberg, et al. A comparison of string distance metrics for name-matching tasks. In *IWeb*, volume 2003, pages 73–78, 2003.
- [3] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *IEEE Transactions on knowledge and data engineering*, 19(1):1–16, 2006.
- [4] M. Theobald, J. Siddharth, and A. Paepcke. Spotsigs: robust and efficient near duplicate detection in large web collections. In *Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 563–570. ACM, 2008.