Alexander Vielhauer
Prof. Dr. Thorsten Papenbrock
AG Big Data Analytics

31.08.–01.09.2023

Programmierpraktikum

Sommersemester 2023

In diesem Block geht es um Datenreinigung, genauer: um das Erkennen und Auflösen doppelter Einträge (Duplikate) in einer Datenbanktabelle. Als Beispieldaten dient uns eine Tabelle, die Restaurants namentlich und mit Adresse listet - ungünstigerweise tauchen einige Restaurants, mit leichten Abweichungen in Name oder Adresse, doppelt auf. Unser Ziel ist es, diese doppelten Einträge zu identifizieren.

Tabelle 1: Restaurant-Einträge mit Duplikaten

id	name	address	city	phone	type
1	arnie morton's	435 s. la cienega blv.	los angeles	310/246-1501	american
2	arnie morton's	435 s. la cienega blvd.	los angeles	310-246-1501	steakhouses
3	art's delicatessen	12224 ventura blvd.	studio city	818/762-1221	american
4	art's deli	12224 ventura blvd.	studio city	818-762-1221	delis
5	hotel bel-air	701 stone canyon rd.	bel air	310/472-1211	californian

Duplikate sind oft nicht genau gleich, sondern unterscheiden sich zum Beispiel durch Schreibweise (Sprache, Tippfehler, Abkürzungen) oder fehlende Werte (siehe Tabelle 1). Daher nutzt man *Ähnlichkeitsmaße*, die zu erfassen versuchen, wie ähnlich zwei Einträge sind. Anschließend werden alle Paare von Einträgen mit ausreichend hoher Ähnlichkeit als Duplikate gekennzeichnet. Das Bereinigen der Duplikate kann dann auf unterschiedliche Weise erfolgen: die doppelten Einträge können fusioniert werden oder es wird einer der beiden Einträge gelöscht.

Eure Aufgabe ist zunächst eine naive Duplikaterkennung zu programmieren (Aufgabe 1). Anschließend geht es darum, zwei *String-Ähnlichkeitsmaße* zu implementieren (Aufgabe 2, 3) und diese zu einem *Record-Ähnlichkeitsmaß* zu kombinieren (Aufgabe 4). Letzteres erlaubt euch ganze Zeilen (Records) im gegebenen Datensatz miteinander zu vergleichen und so Duplikate zu identifizieren. Anschließend gilt es zwei unterschiedliche Vorverarbeitungsschritte zu implementieren, die die Anzahl der zu vergleichenden Record-Paare deutlich reduzieren (Aufgabe 5, 6) und abschließend bleibt die Möglichkeit beliebige Optimierungsversuche zu unternehmen (Aufgabe 7).

Unter `PPClean_Vorlesungsfolien.pdf` findet ihr relevante Folien aus der Vorlesung *Datenintegration*. Die Folien zeigen Beispiele zu den Algorithmen und geben Interessierten noch etwas mehr Kontextinformation. So findet ihr ein motivierendes Beispiel für Duplikaterkennung auf den Folien 2–5.

Bestehensgrenze: 3 Punkte
Technische Voraussetzungen: Java Version ≥ 11
Empfohlen: integrierte Entwicklungsumgebung (z.B. IntelliJ oder Eclipse)

Ein Grundgerüst des zu implementierenden Programms findet ihr in Ilias. Nachfolgend eine kurze Übersicht der relevanten Dateien und Klassen:

```
data
  input.tsv: Restaurant-Datensatz
  ground_truth.tsv: Goldstandard der Restaurant-Duplikate
src
  PPClean
    Data: Datenstrukturen für Tabellen, Records und Duplikate
    DuplicateDetection: Algorithmen zur Erkennung von Duplikaten
    Performance: Bewertung gefundener Duplikate gegen Goldstandard
    Similarity: String- und Record-Ähnlichkeitsmaße
    Configuration.java: Konfigurationsdatei
    Helper.java: Hilfsmethoden
    PPClean.java: main()-Methode für eigene Experimente
  Test
    Task1: Testcode für Aufgabe 1
    ...
    Task6: Testcode für Aufgabe 6
```

Die nachfolgenden Aufgaben 1 bis 6 sind so zu lösen, dass die entsprechenden Tests im Paket `src.Test` erfolgreich durchlaufen. **Schaut euch also die relevanten Klassen und auch den relevanten Testcode an um die Aufgaben zu lösen.**

Als Evaluationsmetriken für die Duplikaterkennung werden in der Regel die Maße *Precision* (Genauigkeit), *Recall* (Trefferquote) und *F1-Score* verwendet. Letzteres ist das harmonische Mittel aus Precision und Recall. Precision und Recall werden ermittelt, indem alle erkannten Duplikate gegen einen Goldstandard (tatsächliche Duplikate) geprüft und gegebenenfalls als falsch erkannte Duplikate gekennzeichnet werden. Auch die nicht erkannten Duplikate im Goldstandard werden als Fehler der Duplikaterkennung notiert und beeinflussen den Recall. Wir ersparen an dieser Stelle die genaue Berechnung; es reicht aus zu wissen, dass Recall, Precision und F1-Score immer zwischen 0 und 1 liegen und ein hoher Wert für erfolgreiche Duplikaterkennung spricht.

Hinweis: Bei Problemen mit dem Lesen und Schreiben von Dateien kann es ggf. helfen, in der Klasse ***`Configuration`*** *absolute Dateipfade anzugeben. Gegebenenfalls müsst ihr in IntelliJ unter* ***`File > Project Structure`*** *noch eine SDK setzen und bei* ***`Language level`*** *"SDK default" auswählen.*

Aufgabe 1: Naive Duplikaterkennung (1 P)

Zuerst wollen wir eine einfache Heuristik zur Duplikaterkennung verwenden: Records beschreiben das gleiche Restaurant genau dann wenn ihre Werte für das Attribut `name` gleich sind.

1. Implementiert die Methode `compare` der Klasse `Similarity.SingleAttributeEquality` um den Test `Task1.testSingleAttributeEquality` zu bestehen. Zwei Records `r1` und `r2` sollen auf Gleichheit an der Stelle `attributeIndex` (im Konstruktor von `SingleAttributeEquality` gesetzt) verglichen werden. Bei Gleichheit soll `1` zurückgegeben werden, bei Ungleichheit `0`. *Tipp: Schaut euch die Klasse* ***`Data.Record`*** *an, um herauszufinden, wie ihr auf den gewünschten Wert zugreifen könnt.*
2. Implementiert die Methode `detect` der Klasse `DuplicateDetection.NaiveDetection` um den Test `Task1.testNaiveDetection` zu bestehen. Gegeben eine `Table` und eine `RecordSimilarity` (z.B. `SingleAttributeEquality`), soll `NaiveDetection` eine Menge an erkannten Duplikaten (`Data.Duplicate`) ausgeben. In der naiven Duplikaterkennung soll jeder Record der Tabelle mit jedem anderen verglichen werden. Zum Vergleich nutzt ihr nun die zuvor implementierte `compare`-Methode. *Tipp: Inkrementiert die Variable* ***`numComparisons`*** *bei jedem Vergleich (Aufruf von* ***`compare`****), um später unterschiedliche Duplikaterkennungsstrategien zu vergleichen.*

Als nächstes gilt es zwei String-Ähnlichkeitsmaße zu implementieren, um danach ein flexibleres Record-Ähnlichkeitsmaß umsetzen zu können.

Aufgabe 2: Jaccard String-Ähnlichkeit (1 P)

Generell ist die Jaccard Ähnlichkeit ein Maß, um die Ähnlichkeit zweier Mengen A und B zu bestimmen. Ihre generelle Formel lautet:

$$Jaccard(A, B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|} \tag{1}$$

Bei der Jaccard String-Ähnlichkeit (`Similarity.Jaccard`) werden die zwei zu vergleichenden Strings `x` und `y` zunächst in `n`-große Substrings zerlegt, welche in den Sets `ngramsX` und `ngramsY` gespeichert werden. Hier zwei Beispiele:

- ngramsX(x="morning", n=3) = {"mor", "orn", "rni", "nin", "ing"}
- ngramsY(y="Hallo Welt", n=4) = {"Hall", "allo", "llo ", "lo W", "o We", " Wel", "Welt"}

Hinweis: Ein String der Länge $|x|$ wird in $|x| - n + 1$ viele Substrings zerlegt.

Anschließend muss herausgefunden werden, wie viele Substrings von `x` auch in den Substrings von `y` vorkommen (also $|$`ngramsX` $\cap$ `ngramsY`$|$). Die zurückzugebende Jaccard-Ähnlichkeit berechnet sich dann wie eingangs generell beschrieben.

Implementiert die `compare`-Methode in `Similarity.Jaccard` um den Test `Task2.testJaccardSimilarity` zu bestehen. *Tipp: Die Länge* ***`n`*** *der zu generierenden Substrings wird im Konstruktor gesetzt. Zusatzinformationen findet ihr auf den Folien 6 und 7.*

Aufgabe 3: Levenshtein String-Ähnlichkeit (1 P)

Die Levenshtein String-Ähnlichkeit basiert auf der Levenshtein-Distanz, auch Editierdistanz genannt, bei der es darum geht, die Anzahl der Zeichenoperationen (Einfügen, Löschen, Ersetzen) zu erfassen, die benötigt werden um einen String in einen anderen zu überführen. Hier zwei Beispiele:

- Levenshtein-Distanz("Tier", "Tor") = 2 (`i` löschen, `e` durch `o` ersetzen)
- Levenshtein-Distanz("Marburg", "Marburger") = 2 (`e` einfügen, `r` einfügen)

Die Levenshtein-Distanz (LD) zweier Strings $x = x_1, x_2, ..., x_m$ und $y = y_1, y_2, ..., y_n$ lässt sich rekursiv wie folgt berechnen:

(Basisfälle)

$$LD(x,y) = \begin{cases} 0, & \text{if } |x| = |y| = 0 \\ |x|, & \text{if } |y| = 0 \\ |y|, & \text{if } |x| = 0 \end{cases} \tag{2}$$

(Rekursionsfälle, $|x| > 0, |y| > 0$)

$$LD(x,y) = \begin{cases} LD((x_1,...,x_{m-1}),(y_1,...,y_{n-1})), & \text{if } x_m = y_n \\ 1 + min \begin{cases} LD((x_1,...,x_{m-1}),(y_1,...,y_{n-1})) \\ LD((x_1,...,x_{m-1}),(y_1,...,y_n)) \\ LD((x_1,...,x_m),(y_1,...,y_{n-1})) \end{cases} & \text{else} \end{cases} \tag{3}$$

Das Paradigma der *dynamischen Programmierung* ist gut geeignet, eine solche rekursive Funktion effizient zu implementieren – denn bei der dynamischen Programmierung geht es um die Zerlegung eines Problems in Teilprobleme. Die Lösungen der Teilprobleme werden gespeichert und zum Lösen der nächstgrößeren (Teil-)Probleme verwandt.

Implementiert die `compare`-Methode in `Similarity.Levenshtein`, indem ihr zunächst die Levenshtein-Distanz der Strings `x` und `y` berechnet. Nutzt zur Speicherung der Teillösungen eine Matrix (z.B. ein zweidimensionales Integer-Array). Intuitiv soll die Matrix von den Basisfällen ausgehend gefüllt werden, bis schließlich das Endergebnis ($LD(x,y)$) ganz unten rechts in der Matrix steht. Konkret sind folgende vier Schritte umzusetzen:

1. Eine Matrix D der Größe $(m+1) \times (n+1)$ erstellen.
2. Die erste Spalte und Zeile mit den Basisfällen errechnen ($D_{i,0} = i$ und $D_{0,j} = j$, für $m \geq i \geq 0, n \geq j \geq 0$).
3. Rekursiv die Zellen $D_{i>0,j>0}$ errechnen.
4. Die Levenshtein-Distanz $LD(x,y)$ steht nun in Zelle $D_{m,n}$.

Als Rückgabewert erwarten wir von der `compare`-Methode jedoch eine Ähnlichkeitsangabe und keine Distanz. Daher gilt es noch die Distanz wie folgt in die Levenshtein-Ähnlichkeit zu überführen:

$$LevenshteinAehnlichkeit(x, y) = 1 - (LD(x, y)/max(m, n)) \tag{4}$$

Abschließend solltet ihr den Test `Task3.testLevenshteinSimilarity` bestehen. *Tipp: Implementiert die oben genannten vier Schritte einzeln und nacheinander. Nutzt die* `charAt`*-Methode auf Strings um einzelne Buchstaben zu vergleichen. Zusatzinformationen findet ihr auf den Folien 8–12.*

Aufgabe 4: Hybride Record-Ähnlichkeit (1 P)

Um nun ganze Records umfangreich vergleichen zu können, kombinieren wir die String-Ähnlichkeitsmaße Levenshtein und Jaccard zu einem *hybriden* Record-Ähnlichkeitsmaß. Im Konstruktor von `Similarity.Hybrid` werden die beiden String-Maße instanziiert und es wird eine Liste an `policies` gesetzt: Jeder Eintrag in der Liste beschreibt wie die Werte der Records an der entsprechenden Position verglichen werden sollen. `[null, "L", "J", "L"]` würde beispielsweise das erste Attribut ignorieren, das dritte mittels Jaccard String-Ähnlichkeit vergleichen und das zweite und vierte Attribut mit Levenshtein-Ähnlichkeit.

Eure Aufgabe ist es nun, die `compare`-Methode in `Similarity.Hybrid` zu implementieren und die Tests in `Task4` zu bestehen. Als Rückgabewert wird die durchschnittliche Ähnlichkeit aller verglichenen Attribute erwartet. Gegebenenfalls müsst ihr eure Implementierung von `DuplicateDetection.NaiveDetection.detect` anpassen: Die `Task4` Tests erwarten nun, dass Paare von Records als Duplikate gelten sobald sie eine gewisse durchschnittliche Ähnlichkeit überschreiten. Diese Grenze nennen wir auch `threshold` und wir setzen sie im Konstruktor der `NaiveDetection` Klasse.

Aufgabe 5: Sorted-Neighborhood (1 P)

Da die naive Strategie Records miteinander zu vergleichen quadratische Laufzeit hat (jeder Record muss mit jedem anderen verglichen werden), geht es in dieser Aufgabe darum einen Vorverarbeitungsschritt namens *Blocking* zu implementieren. Dabei werden Duplikatkandidaten zu überschaubar großen Blöcken zusammengefasst, deren Mitglieder man anschließend wieder mittels Ähnlichkeitsmaßen auf Duplikate vergleicht.

Die *Sorted-Neighborhood*-Methode ist ein solches Blocking-Verfahren und funktioniert in drei Schritten:

1. Für jeden Record einen *key* (Schlüssel) berechnen.
2. Die Tabelle nach Schlüsseln sortieren.
3. Ein "Fenster über die Tabelle schieben", sodass Records immer nur mit ein paar wenigen anderen Records (denen im gleichen Fenster) verglichen werden.

Die Berechnungsvorschrift zur Schlüsselgenerierung und die Größe des Fensters sind entscheidende Parameter dieser Methode; wir setzen sie neben dem `threshold` im Konstruktor von `Du-`

`plicateDetection.SortedNeighborhoodDetection`. Eine Liste namens `keyComponents` gibt für jede Position an, wie viele Zeichen des entsprechenden Wertes zur Schlüsselgenerierung verwendet werden sollen. `[0, 3, 1, 3, 0, 0]` würde zum Beispiel für die fünf Records aus Tabelle 1 die Schlüssel "arn4los", "arn4los", "art1stu", "art1stu" und "hot7bel" erzeugen. Wenn wir lexikographisch nach diesen Schlüsseln mit einer Fenstergröße von 3 sortieren, dann würde erst der Eintrag #1 mit den Einträgen #2 und #3 verglichen werden, dann der Eintrag #2 mit #3 und #4 und abschließend Eintrag #3 mit Einträgen #4 und #5.

Implementiert die Sorted-Neighborhood-Methode, indem ihr die `detect`-Methode in `DuplicateDetection.SortedNeighborhoodDetection` programmiert. Dafür müsst ihr zunächst die Schlüsselgenerierung in `Record.generateKey` und die Tabellensortierung nach Schlüsseln in `Table.sortByKey` implementieren. *Tipp: Inkrementiert die Variable `numComparisons` bei jedem Vergleich (Aufruf von `compare`), um euch mit der naiven Duplikaterkennung (`NaiveDetection`) zu vergleichen. Zusatzinformationen findet ihr auf den Folien 13–17.*

Aufgabe 6: Locality Sensitive Hashing (1 P)

Knobelaufgabe

Ein alternatives Vorgehen zum Blocking ist *Locality Sensitive Hashing (LSH)*. Anders als beim normalen Hashing—bei dem es Hash-Kollisionen zu vermeiden gilt—geht es beim LSH darum ähnlichen Werten den gleichen Hashwert zu geben, sie also in den gleichen hash bucket zu legen.

Das gelingt uns indem wir für jeden Record eine Integer-Signatur erstellen und dann immer nur kurze Teile dieser Signatur hashen. Sind zwei Records in mindestens einem Teil ihrer Signatur gleich, landen sie im gleichen hash bucket und wir nehmen sie in die Liste der Duplikatkandidaten auf.

Implementiert das Blocking mit Locality Sensitive Hashing, indem ihr die `detect`-Methode in `DuplicateDetection.LSHDetection` programmiert. Dazu müsst ihr zunächst die drei Methoden `calculate...` implementieren, die `detect` aufruft.

- Tokenization (`calculateTokens`): Zunächst müssen wir das `tokenUniverse` und die `tokenMatrix` bestimmen. Ähnlich wie bei der Jaccard-Ähnlichkeit in Aufgabe 2 zerlegen wir Strings in gleichgroße Substrings (tokens), hier der Größe `tokenSize`. Das `tokenUniverse` ist eine Liste (ohne Duplikate) aller tokens die in der Tabelle vorkommen. Die `tokenMatrix` hat eine Reihe pro Token (i-te Reihe entspricht i-tem token im `tokenUniverse`) und eine Spalte pro Record. Die Zelle (i,j) ist mit `true` belegt falls das i-te token (im `tokenUniverse`) im j-ten Record vorkommt. *Hinweis: Nutzt `Record.toString()` um für einen Record einen String zu bekommen, den ihr dann tokenisiert.*
- MinHashing (`calculateMinHashes`): Um für jeden Record eine Signatur zu erstellen gibt es die `signatureMatrix`. Die Länge der Signatur ist bestimmt durch `numMinHashs` und bestimmt die Anzahl der Reihen der Matrix. Die Anzahl der Spalten der Matrix ist gleich der Anzahl der Records in der Tabelle. Für die erste Reihe der Matrix (die erste Stelle der Signatur) gilt es für jeden Record j in der entsprechenden Spalte in der `tokenMatrix` die Zelle (k, j) zu finden in der `true` steht und k minimal ist. Es geht also darum für jeden Record den Reihenindex des ersten vorkommenden tokens zu finden. Die Zelle (0, j) der

`signatureMatrix` wird dann auf k gesetzt. Sprich: Der erste Wert einer Signatur ist der Reihenindex des ersten vorkommenden tokens. Das "erste token" meint hier die Reihenfolge in der `tokenMatrix`, nicht das erste token im ursprünglichen String. Um die weiteren Signaturwerte zu berechnen, permutiert (shuffled) man die Reihen der `tokenMatrix` und wiederholt das eben beschriebene Vorgehen. *Hinweis: Nutzt* ***`Helper.shuffleMatrix()`*** *um eine Permutation der* ***`tokenMatrix`*** *zu bekommen.*

- LSH (`calculateHashBuckets`): Nun zerlegen wir die Signaturen der Records in *bands*, `numBands` gibt uns an in wie viele gleichgroße bands wir die Signaturen zerlegen sollen. Pro Teilstück der Signaturen erstellen wir eine Hashtabelle, die Integer-Hashwerte auf Listen von Record-IDs (hash buckets) abbildet. Nutzt die Methode `hash` um den Hashwert für ein band einer Signatur zu erhalten. *Hinweis: Ihr könnt annehmen, dass sich die Signatur restlos in gleichgroße bands zerlegen lässt.*

Nun könnt ihr in der `detect`-Methode über alle Hashtabellen und innerhalb einer Hashtabelle über die belegten Schlüssel iterieren, um alle Records mit den Record-IDs aus den zugehörigen hash buckets mit der Recordähnlichkeit `recSim` zu vergleichen.

Tipp: Inkrementiert die Variable ***`numComparisons`*** *bei jedem Vergleich (Aufruf von* ***`compare`****), um euch mit der naiven Duplikaterkennung (*`NaiveDetection`*) zu vergleichen. Zusatzinformationen findet ihr auf den Folien 18–31.*

Aufgabe 7: Optimierung (1 P)

Schon fertig?! Dann überlegt doch mal, wie ihr eure Duplikaterkennung weiter verbessern könnt. Schafft ihr es, die in den Tests erzielten Precision, Recall und F1-Scores zu übertreffen? In der Klasse `PPClean` müsst ihr euch dafür analog zu den Tests ein Reinigungs-Scenario zusammenbauen. Ideen zur Optimierung sind dann folgende:

1. **Parametrisierung:** Probiert verschiedene Fenstergrößen und Sortierungsschlüssel für die Sorted Neighborhood Methode aus. Auch eine Variation des zusammengesetzten Ähnlichkeitsmaßes und die Optimierung des Ähnlichkeits-Thresholds kann Verbesserungen bringen.
2. **Ähnlichkeitsmaße:** Neben *Jaccard* und *Levenshtein* gibt es noch eine Vielzahl weiterer Ähnlichkeitsmaße. Probiert doch mal *Soundex*, *Jaro-Winkler* oder *Monge-Elkan*. Diese gibt es auch vorimplementiert in Java Bibliotheken wie Apache Commons Text[1] oder Apache Commons Codex[2].
3. **Duplicate Detection:** Die Sorted Neighborhood Methode ist nur eines von vielen Blocking Verfahren. Popular sind auch Hashing-basierte Ansätze. Implementiert doch versuchsweise einmal einen neuen `DuplicateDetection` Ansatz mittels Hashing.
4. **Transitive Hülle:** Fehlende Duplikate können nach der Suche auch durch die Berechnung

[1] `https://commons.apache.org/proper/commons-text/javadocs/api-release/org/apache/commons/text/similarity/package-summary.html`

[2] `https://commons.apache.org/proper/commons-codec/apidocs/org/apache/commons/codec/language/package-summary.html`

der transitiven Hülle noch gefunden werden. Habt ihr (A,B) und (B,C) als Duplikate gefunden, dann muss auch (A,C) ein Duplikat sein. Berechnet all diese fehlenden Duplikate!

5. **Parallelisierung:** Die Ähnlichkeitsberechnungen in der Duplikaterkennung sind alle unabhängig voneinander und ihre Ausführungen sind daher super leicht zu parallelisieren. Für die Parallelisierung könntet ihr Java ThreadPools[3] oder parallele Streams[4] einsetzen.

Aufgabe 8: Werbung

Für weitere Lehre zu Datenbanken und Datenmanagement besucht gerne unsere Vorlesung *Datenbanksysteme* im SoSe24. Für Masterstudentinnen und -studenten bieten wir auch die Vorlesungen *Datenintegration* und *Big Data Systems* an. Aktuelle Informationen zu Lehre und Abschlussarbeiten findet ihr stets auf unserer Website[5].

[3] `https://www.baeldung.com/thread-pool-java-and-guava`
[4] `https://www.baeldung.com/java-when-to-use-parallel-stream`
[5] `https://www.uni-marburg.de/en/fb12/research-groups/big_data_analytics`