Bioinformatikseminar 2007/2008

Restricted Random Walk Clustering

Christian Breitenberger

Einleitung

Motivation

Warum sollte man Datensätze versuchen wollen zu gruppieren? Dafür kann es mehrere Gründe geben und ein paar davon möchte ich hier vorstellen:

- Büchereien: Zum automatisierten Erstellen von Index–und Suchbegriffen
- Büchereien: Zum finden und anzeigen von alternativen Vorschlägen einer Büchersuche

Dies sind nur einige Beispiele, warum es Sinn macht, Datensätze gruppieren zu wollen.
Ein Beispiel eines solchen Algorithmus ist das Restricted Random Walk Clustering

Restricted Random Walk Clustering

Graph

Als erstes wollen wir einen Graphen für den obigen Algorithmus definieren:

Definition

Es sei $G(V, E, \omega)$ ein Graph mit V Knoten, E Kanten zwischen zwei Knoten mit positivem Kantengewicht und $\omega_{i,j}$ die Ähnlichkeit (oder die metrische Entfernung) zwischen zwei Knoten i, j.

Walks

Nun kann man folgenden Algorithmus definieren:

Definition

1. Wähle einen beliebigen Startknoten $n_0 \in V$
2. Konstruiere T mit $T = V \backslash \{n_0\}$
3. Wähle beliebigen Nachfolgerknoten $n_1 \in T$ und merke sich ω_{n_0,n_1}
4. Konstruiere T' mit $T' = T \backslash V'$ mit $V' =$ Menge der Knoten, deren Ähnlichkeit kleiner ist als ω_{n_0,n_1}
5. Wähle eine beliebigen Nachfolger zu n_1 aus T'
6. Wiederhole Schritt 2-4 solange bis $T' = \emptyset$
7. Speichere den erhaltenen Walk
8. Wiederhole Schritt 1-7 n–mal für jedes $v \in V$

Clusterkonstruktion

Mit den so erhaltenen Walks kann man sich nun daran machen, die in dem Datensatz enthaltenen Knoten in Cluster zu gruppieren. Dazu gibt es zwei Möglichkeiten:

1. Componentcluster
2. Walk Context Cluster

Cluster: Componentcluster

Folgender Algorithmus wird zu Erstellung der Cluster verwendet

Definition

1. Konstruiere eine Serie von Graphen $G_k = (V, E_k)$ mit E_k eine Verbindung zwischen zwei Knoten, wenn sie im k–ten Schritt eines Walks beteiligt waren
2. Konstruiere aus diesen Graphen eine zweite Serie Graphen $H_k = \cup_{l=k}^{\infty} G_l$
3. Cluster sind die verbundenen Untergraphen von H_k

Componentcluster: Nachteil

Die oben genannte Methode hat eine gravierenden Nachteil, den sogenannten „Brückeneffekt“
Dieser Effekt bewirkt, dass kleinere Cluster mittels nur eines Knotens zu einem großen Cluster verschmolzen werden, obwohl dies unerwünscht ist.
Beispiel: Statistik für Psychologen

Cluster: Walk Context Cluster

Eine andere Methode zur Konstruierung der Cluster ohne unerwünschte Brückeneffekte ist das Walk Context Clustering

Definition

1. Wähle ein Cut–Off–Level k
2. Suche alle Walks, in welchen der gesuchte Knoten nach k auftritt.
3. Ein Cluster sind alle Knoten aller Walks welche nach k auftreten.

Dies hat den Vorteil, dass nicht ganze Walks mit dem Brückenelement verschmolzen werden, sondern nur die Knoten, welche nach k mit Hilfe des Brückenelements erreicht werden können.

Probleme durch Grapheninhomogenität

- Ein Problem kann die Homogenität des Graphens sein. Je nach Graph kann es stark besiedelte Zonen geben, während in anderen Gebieten der Graph nur sehr dünn besiedelt ist.
- Dies führt zu unterschiedlichen Längen eines Walks
- Dies führt dazu, dass gewisse Daten ab einem bestimmten k nicht mehr berücksichtig werden
- Lösung: Einführung eines relativem Maß, dem sogenannten l–Maß

I–Maß

Das I–Maß ist ein relatives Maß für die „Güte“ eines Walks und ist wie folgt definiert

Definition

$$I = \frac{\text{Schrittzahl}}{\text{Walklänge}}$$

Allerdings werden hier Walks der Form 1/1 bzw 10/10 gleich gewichtet, wobei der letztere der beiden deutlich mehr Gewicht haben sollte. Dementsprechend wurden zwei weitere Maße eingeführt, I^+ und I^-.

l^+– und l^-–Maß

Definition

- $l^+ = \frac{\text{Schrittzahl}}{\text{Walklänge}_{+1}}$
- $l^- = \frac{\text{Schrittzahl}_{-1}}{\text{Walklänge}}$

l^+ bestraft kleinere Walks härter als l^- aber beide liefern asymptotisch das selbe Verhalten

Effizienz

Die folgenden Laufzeiten gebe ich nur der Vollständigkeit mit an, ein Beweis entfällt

Theorem

1. *Laufzeit von n Walks:* $O(n \log n)$
2. *Componentcluster:* $O(n^3)$
3. *Walk Context Cluster:* $O(n \log^2 n)$

Updates

Motivation

Da Cluster in den meisten Fällen in dynamischen Umgebungen eingesetzt werden, stellt sich die Frage, wie man auf Updates des Graphens reagieren kann. Eine Möglichkeit wäre es, den bestehenden Algorithmus noch einmal komplett über den aktualisierten Graphen laufen zu lassen. Dies ist aber in Anbetracht großer Datenmengen nicht immer möglich und so stellt sich die Frage nach effizienten Updatealgorithmen, welche das selbe leisten wie der ursprüngliche Algorithmus.

Algorithmen

Grundsätzlich lassen sich 3 Fälle bei einem Update unterscheiden:

1. Aktualisierung der Ähnlichkeiten
2. Einfügen eines Knotens
3. Entfernen eines Knotens

Wie wir später sehen werden, lassen sich Fall 2 und Fall 3 auf den ersten Fall zurückführen. Dementsprechend werden wir uns im folgenden Fall 1 genauer ansehen.

Aktualisierung der Ähnlichkeiten

Eine Aktualisierung kann 3 verschiedene Fälle zur Folge haben:

1. Es wird kein neuer Weg möglich oder unmöglich
2. Ein neuer Weg wird möglich
3. Ein bereits gewählter Weg wird illegal

Von diesen Fällen sind nur Fall 2 und Fall 3 interessant. Fall 1 ist trivial und für die weitere Betrachtung nicht von Relevanz

Neuer Pfad: Knotennmenge

Zuerst werde ich die nötigen Knotenmengen definieren

Definition

- T_e möglicher Nachfolger der Kante e vor der Aktualisierung
- T'_e möglicher Nachfolger der Kante e nach der Aktualisierung

Neuer Pfad: Algorithmus

Mit obiger Definfition kann man nun folgenden Algorithmus konstruieren:

Definition

1. $\forall e' \in E$ mit $e \in T'_{e'}$ und $e \notin T_{e'}$:
2. $\forall e''$ mit $e'' \in T'_{e'}$:
3. $\forall$ walks $(\dots, e', e'', \dots)$
4. Mit Wahrscheinlichkeit $1 - \frac{|T'_e|}{|T'_{e'}|}$ schneide den Walk ab e'' ab.
5. Füge e am ende des Walks ein.
6. Setze den Walk mit T'_e fort.
7. Benutze den Originalalgorithmus für den Rest des Walks.

Neuer Pfad: Eigenschaften

Der Updatealgorithmus bestitzt folgende Eigenschaften, welche ich je nach Zeit näher beweisen werde.

Theorem

Der Updatealgorithmus liefert die selbe Wahrscheinlichkeitsverteilung für die Cluster wie die Ausführung des Originalalgorithums über die aktualisierten Daten

Theorem

Die Dichteverteilung jedes Mitglieds von T'_{ij}, sprich nach Auftreten von neuen Walks ist $B(m, \frac{1}{n})$.

Illegale Pfade

Widmen wir uns jetzt dem Algorithmus für illegale Pfade. Dieser ist wie folgt definiert:

Definition

1. $\forall$ walks $(\ldots, e', e'', \ldots)$
2. Schneide den Walk von e ab.
3. Wähle e'' von einer Gleichverteilung über $T'_{e'}$
4. Füge e'' am Ende des Walks ein
5. Setze den Walk durch Wahl eines Nachfolgers von $T'_{e''}$ fort
6. Benutze den Originalalgorithmus für den Rest des Walks.

Illegaler Pfad: Eigenschaften

Auch hier werde ich die Behauptung je nach Zeit beweisen

Theorem

Die Dichteverteilung jedes Mitglieds von T'_{ij}, sprich nachdem Walks illegal geworden sind, ist $B(m, \frac{1}{n})$.

Knotenupdates

Bsi jetzt haben wir uns nur um die Kanten gekümmert und neue bzw. gelöschte Knoten außer acht gelassen. Dies werde ich jetzt nachholen

1. *Neuer Knoten:* Zuerst wird von dem neuen Knoten aus n–mal der Originalalgorithmus ausgeführt. Dann werden alle Walks, welche den neuen Knoten über Kanten einbeziehen könnten angesehen und mit Hilfe des Updatealgorithmus entschieden, ob bestehende Walks verändert werden müßen.
2. *Löschen von Knoten:* Als erstes werden alle Walks, welche von dem gelöschten Knoten gestartet sind, gelöscht. Danach müßen alle Walks, welche den gelöschten Knoten besucht haben, mittels dem Updatealgorithmus neu bewertet werden.

Hier sei noch erwähnt, dass durch ein Hinzufügen von Knoten keine alten Walks ungültig werden sowie durch das Löschen von Knoten keine neuen Walks geschaffen werden.

Update: Effizienz

Die Komplexität der Updatealgorithmen ist folgendermaßen gegeben:

Theorem

$O(\frac{k \log^2 n}{\bar{d}})$

Beweis.

Es sei $\bar{d}$ der durschnittliche Grad eines Knotens $V \in G$. Daraus folgt das G $\frac{\bar{d}n}{2}$ Kanten besitzt. Jeder Walk besitzt nach Konstruktion eine Länge von $O(\log n)$. Wenn man nun k Walks von jedem der n Knoten aus startet erhält man eine Anzahl von $O(nk \log n)$ Schritten. Daraus folgt, dass jeder Knoten im Durchschnitt von $O(k \log n)$ Walks und jede Kante von $O(\frac{2k \log n}{\bar{d}})$ Walks besucht wird.
Wenn nun ein neuer Nachfolger akzeptabel wird, werden im Mittel $O(k \log n)$ betrachet, von welchen dann $O(\frac{2k \log n}{\bar{d}})$ neu berechnet werden müssen. Da ein Walk eine Länge von $O(\log n)$ besitzt und im Mittel von der Länge her halbiert wird, müssen $O(\frac{1}{2} \log n)$ Schritte gelöscht werden. Wenn die Daten der Walks in linearer Zeit verfügbar sind resultiert dies in einer Laufzeit von $O(\frac{k \log^2 n}{\bar{d}})$
Das hinzufügen von neuen Kanten besitzt den selben Aufwand

Abschluß

Bewertung des Algorithmus

Der in diesem Vortrag vorgestellte Algorithmus arbeitet vor allem auf großen Datensätzen effizient und die Updatealgorithmen sind soweit verbessert, dass sie eine Zeitersparnis erbringen (An dieser Stelle sei erwähnt, dass wir hier von Graphen mit ca. 1 Million Knoten reden). Dies ist auch nötig, denn in der heutigen Zeit ist es in vielen Anwendungen enorm wichtig geworden, schnell und effizient auf Veränderungen in dynamischen Datensätzen reagieren zu können ohne das die Qualität darunter leidet