Institut für Angewandte und Numerische Mathematik
Prof. Dr. Tobias Jahnke • Dipl.-Math. oec. Marcel Mikl

Numerische Mathematik 2

Sommersemester 2015

Programmierblatt 2

Aufgabe 2 (Google's Eigenwertproblem)

Eine alltägliche Situation. Sie geben in die Suchmaschine „google" ein Stichwort ein und erhalten eine Liste von Seiten mit Informationen zu diesem Stichwort. Es ist naheliegend, dass die gelisteten Seiten das Stichwort enthalten müssen, doch woher weiß google, welche dieser Seiten interessant sind und welche nicht? Immerhin sollen unter den Top Ten der Treffer zufriedenstellende Ergebnisse sein.

Die Idee des Random Surfer

Larry Page und Sergey Brin, die Gründer von google und Erfinder des „Page-Ranks", haben sich dazu folgendes überlegt. Wir stellen uns einen Web-Surfer vor. Er hat die Seite i in seinem Webbrowser geöffnet und liest sie. Auf der Seite i gibt es $l_i \in \mathbb{N}_0$ viele Links $j_1, \dots, j_{l_i}$. Einen dieser Links klickt er an und wechselt auf die Seite j. Er liest die Seite j und klickt einen Link an und landet auf der nächsten Seite k. Die Wahl des Links geschieht dabei nach Zufallsprinzip. Wir nennen den Surfer daher Random Surfer.

Die Link-Beziehung zwischen den $n \in \mathbb{N}$ Webseiten beschreibt die Adjazenzmatrix $A \in \{0,1\}^{n,n}$. Wenn von der Seite i ein Link zur Seite j führt, so ist $A_{i,j} = 1$ und andernfalls ist $A_{i,j} = 0$. Selbstverlinkungen wollen wir nicht zulassen, daher fordern wir für die Diagonalelemente $A_{j,j} = 0$. Unser Random Surfer befindet sich auf der Seite i, was wir durch den Vektor $v = (\delta_{i,l})_{l=1}^n$ beschreiben. Er befindet sich mit Wahrscheinlichkeit 1 auf der Seite i. Nun lassen wir ihn einmal klicken. Anschließend befindet er sich auf einer der verlinkten Seiten, also einer Seite j so, dass $(A^T v)_j = 1$ gilt. Suchen wir die relative Wahrscheinlichkeit für das Aufrufen der Seite j, so müssen wir eine Normalisierung der Zeilen von A durchführen.

$$H_{i,j} := \begin{cases} \frac{A_{i,j}}{l_i} & \text{falls } l_i := \sum_{l=1}^n A_{i,l} > 0 \\ 0 & \text{sonst} \end{cases} \tag{1}$$

und erhalten die „Hyperlink-Matrix" H.

Unser Random Surfer startet auf einer beliebigen Seite und nach k Klicks befindet er sich mit der Wahrscheinlichkeit $((H^T)^k v)_j$ auf der Seite j. Führen wir dies mit der Hyperlink Matrix des World Wide Web durch, so erhalten wir

$$\lim_{k\to\infty} (H^T)^k v = 0 \tag{2}$$

d.h. unser Random Surfer befindet sich letztlich auf keiner Seite. Was ist da passiert? Es gibt Webseiten, die keinen Link enthalten. Surft unser Random Surfer solch eine Seite – „dangling node"genannt – an, so kann er sich nicht mehr weiter klicken. Der nächste Klick ist damit das Schließen seines Webbrowsers.

Erste Modellerweiterung

Das Problem (2) haben Page und Brin sehr elegant gelöst. Was macht ein realer Surfer der auf einer solchen Seite landet? Er kann sich zwar nicht weiterklicken, wird allerdings auf andere Weise eine neue Seite aufrufen. Welche das sein wird, ist schwer zu sagen. Generell ist jede Seite gleichermaßen wahrscheinlich. Wir haben also die Nullzeilen in H durch $\frac{1}{n}$-Zeilen zu ersetzen. Für eine saubere Schreibweise definieren wir dazu Vektoren $a = (a_i)$ und $e = (e_i) \in \mathbb{R}^n$ mit $e_i = 1$ für $i = 1, \dots, n$ und

$$a_i = \begin{cases} 1 & \text{Seite } i \text{ enthält keine Links bzw. } A_{i,:} = (0, \dots, 0) \\ 0 & \text{sonst} \end{cases} \tag{3}$$

und

$$S := H + a\left(\frac{1}{n} e^T\right) \tag{4}$$

S ist also die Matrix, die das Problem der dangling nodes löst.

Wir lassen den Random Surfer erneut unendlich oft klicken. Es gilt

$$\lim_{k\to\infty} \|(S^T)^k v\|_1 = \lim_{k\to\infty} e^T (S^T)^k v = 1. \tag{5}$$

Das Ergebnis ist gut, denn der Random Surfer geht weder verloren, noch vervielfältigt er sich.

Die Suchmaschine braucht für jede Seite den Page-Rank, was nichts anderes ist, als die relative Häufigkeit, mit der unser Random Surfer die jeweilige Seite besucht hat, also $p = \lim_{k\to\infty} (S^T)^k v$. Die Idee dahinter ist naheliegend. Seiten mit qualitativ gutem Inhalt, werden von vielen anderen Seiten verlinkt. Schlechte Seiten werden dagegen selten verlinkt. Somit besucht unser Random Surfer die hochwertigen Seiten häufiger als die minderwertigen. Nun müssen wir nur noch sicherstellen, dass der Random Surfer auch jede Seite besuchen kann egal von auf welcher Seite er sich gerade befindet.

Zweite Modellerweiterung

Stellen wir uns folgendes Szenario vor. Es gibt 7 Webseiten $w_1, \ldots, w_7$. Die Seiten w_1, w_2 und w_3 verlinken sich untereinander, aber nicht zu den w_4, w_5, w_6. Die Seiten $w_4, \ldots, w_7$ verlinken sich wiederum untereinander, aber nicht zu den w_1, w_2, w_3. Der Webgraph zerfällt in disjunkte Teilgraphen. Beginnt unser Random Surfer in einem dieser beiden Teilgraphen, so wird er den anderen Teilgraphen nie erreichen. Page und Brin haben sich dazu gedacht: kein Mensch verfolgt dauerhaft die Links der Webseiten. Nach einer gewissen Zeit wird der Surfer eine neue Adresse in seinen Browser eingeben und beginnt dort für eine gewisse Zeit die Links zu verfolgen. Die Eingabe einer neuen Website hatten wir schon bei den dangling nodes eingeführt. Diesmal kann der Surfer auch eine Seite verlassen, die Links enthält. Diese Eigenschaften besitzt die „Google-Matrix“

$$G_\alpha = \alpha S + (1-\alpha)\frac{1}{n} e e^T \tag{6}$$

Der Parameter α steuert wie stark die Links verfolgt werden gegenüber zufälligem Ansurfen einer beliebigen Seite. Sinnvoll ist $\alpha \in [0.5, 1)$. Die Google-Matrix G_α stellt sicher, dass die Berechnung des Page-Ranks mit den gewünschten Eigenschaften funktioniert. Die mathematische Herleitung und die Beweise finden Sie in [1]. Der Page-Rank ist

$$p_\alpha := \lim_{k\to\infty} (G_\alpha^T)^k v \tag{7}$$

wobei $v \in [0,1]^n$ ein beliebiger Vektor mit $e^T v = 1$ ist. Dass dies funktioniert und weitere mathematische Eigenschaften der Google Matrix G sind in [1] ausführlich und gut verständlich dargestellt und bewiesen.

Technische Schwierigkeiten

Wir nehmen an, dass die Adjazenzmatirx A bereits bekannt ist. Bei Google ist n in den Billionen. Wenn wir alle Einträge von A speichern wollen, so sind es n^2 viele. Das Speichermedium dafür gibt es noch nicht. Erinnern wir uns, dass die Adjazenzmatrix nur aus Nullen und Einsen besteht. Die Anzahl der Einser in jeder Zeile ist gleich der Anzahl der Links auf der zugehörigen Seite, also relativ klein. Im Durchschnitt stehen in jeder Zeile nur ca. 10 von Null verschiedene Einträge. Daher speichert Google für A zu jeder Zeile die Positionen der Eins-Elemente ab. Das kostet dann $10n$ an Speicherplatz.

Dieses spezielle Speicherformat für „dünn besetzte“ Matrizen (englisch „sparse“) ist in MatLab bereits enthalten. Die Notation bei den Grundrechenarten erfolgt wie Sie es gewohnt sind. Siehe auch `help sparfun`. Das Gegenteil zu dünn besetzten Matrizen sind „voll besetzte“ Matrizen („dense“). Mit diesen haben Sie bisher gearbeitet.

Bei der Matrix H können wir analog vorgehen. S schreiben wir besser nicht als eine Matrix, denn dort gibt es Zeilen die komplett von Null verschieden sind und uns den Speicherplatz auf n^2 erhöhen. Die Schreibweise mit den zwei Summanden in (4) kostet uns dagegen nur $\mathcal{O}(n)$ an Speicherplatz. Analog verhält es sich mit der Google-Matrix G.

Aufgabenstellung

Sie sollen sich nun überlegen, wie Sie den Page–Rank p_α in endlicher Zeit auf einem Computer berechnen können. Ausgehend von (7) kann man zeigen, dass

$$G_\alpha^T p_\alpha = p_\alpha$$

gelten muss. Diesen Typ von Gleichung haben Sie in der Vorlesung kennen gelernt.

1. Überlegen Sie sich, wie sie für festes $\alpha \in (0,1)$ und einen gegebenen Vektor v das Produkt $(G_\alpha)^T v$ effizient bezüglich Speicherbedarf und Rechenzeit ausrechnen können. Beachten Sie insbesondere den Aspekt, dass die Rechnung für viele Vektoren v wiederholt wird.
2. Wir interessieren uns für den Rang bestimmter Seiten, also den Page–Rank p_α. Implementieren Sie dazu eine effiziente Funktion

   ```
   p = page_rank_power(A, alpha, tol)
   ```

   die den Eigenvektor p von G_α zum größten Eigenwert 1 mit der Potenzen-Methode berechnet. Brechen Sie die Iteration ab, sobald die relative Änderung des genäherten Eigenvektors in der 1-Norm

   $$\texttt{err}_k = \frac{\|v_{k+1} - v_k\|_1}{\|v_k\|_1}$$

   kleiner als `tol` ist.
3. Laden Sie sich die Dateien von der Vorlesungs-Webseite herunter. Das Zip-Archiv enthält die Adjazenzmatrix und die Liste der Seitenadressen (URL) verschiedener Webseiten. Eine Funktion zum Einlesen der Datensätze ist ebenfalls dabei. Wenden Sie Ihre Routine auf diese Beispiele an. Gute Werte für diese Tests sind z.B. $\alpha = 0.8$ und $\texttt{tol} = 10^{-5}$.

4. Versuchen Sie experimentell die Konvergenzrate der Potenzenmethode im Beispiel zu bestimmen. Überlegen Sie sich dazu zuerst

$$\texttt{err}_k \approx \|v_k - p_\alpha\| \quad \text{und} \quad \log(\texttt{err}_k) \approx \log(C) + k\log(\eta), \quad \eta = \frac{\lambda_2}{\lambda_1}$$

also warum das Abbruchkriterium als Fehlerschätzung verwendet werden kann. Erweitern Sie Ihr Potenzenmethode so, dass $\texttt{err}_k$ mit ausgegeben wird und finden Sie mit linearer Ausgleichsrechnung und den Ansatzfunktionen $\varphi_0(t) = 1$, $\varphi_1(t) = t$ die Größen $C > 0$ und $\eta \in (0, 1)$. Visualisieren Sie sowohl den Fehler als auch die geschätzte Konvergenzrate.
Hinweis: Ignorieren Sie für die Bestimmung der Konvergenzrate die erste Hälfte der Fehlerschätzungen, da sonst das Verhalten für kleine k die approximative Konvergenzrate verfälscht.

5. Vergleichen Sie nun die Geschwindigkeit ihrer Routine mit Matlabs eingebauter Funktion `eigs`. Schreiben Sie dazu eine Funktion

`p = page_rank_eigs(`A`, `α`, tol)`

und übergeben Sie der Funktion `eigs` anstelle der Google-Matrix einen Funktions-Handle
(`help function_handle`), über den die Matrix-Vektor-Multiplikation so effizient wie bei Ihrer Potenzenmethode ausgeführt wird.
Hinweis: Matlab stellt zur Messung der CPU-Zeit die Funktionen `cputime` und `tic/toc` zur Verfügung.

Literatur

[1] Amy N. Langville and Carl D. Meyer, *Google's PageRank and Beyond*, Princeton University Press (2006)

Die Aufgabe kann von **Dienstag, den 9. Juni 2015, 14:00 Uhr** und
Mittwoch, den 10. Juni 2015 , 15:45 Uhr an in den Programmiertutorien bearbeitet werden.

Homepage:

Unter http://www.math.kit.edu/ianm3/lehre/numa022015s/de erreichen Sie die Homepage zur Vorlesung. Dort finden Sie neben den aktuellen Übungsblättern auch alle Informationen zum Vorlesungsbetrieb.