Performanzoptimierung, Einsatz und Evaluation des C++ Programms *EOS_V2* zur Satzendeerkennung

Bachelorarbeit Stefan Schweter (WS 2013/2014) - Betreuer:
Dr. Maximilian Hadersbeck

Kolloquium und Repetitorium - 28. April 2014

Überblick

- Satz und Satzgrenze
- *EOS_V2* - das Programm zur Satzendeerkennung
- Performanzoptimierung
- Coding Style und Unittests
- Evaluation
- Fazit und Ausblick

Definition eines Satzes (1)

- Duden-Definitionen[1]:

  Ein Satz ist eine abgeschlossene Einheit, die nach den Regeln der Syntax gebildet worden ist.

  Ein Satz ist die größte Einheit, die man mit den Regeln der Syntax erzeugen kann.

  Ein Satz ist eine Einheit, die aus einem finiten Verb und allen vom Verb verlangten Satzgliedern besteht.

[1] Siehe `http://de.wikipedia.org/wiki/Satz_%28Grammatik%29`

Definition eines Satzes (2)

- Karl Bühler[1]:

  Sätze sind die einfach selbständigen, in sich abgeschlossen Leistungseinheiten oder kurz die Sinneinheiten der Rede.

- Hanz Glinz[2]:

  Ein Satz ist eine pragmatische Einheit, die im geschriebenen/gelesenen Text mit einem Großbuchstaben anfängt und durch Punkt, Ausrufezeichen oder Fragezeichen abgeschlossen ist (wobei man nachher wieder gross weiterfährt)

[1] Zitat wörtlich entnommen aus Dürr (2006), Seite 271
[2] Ebenda.

Satz und Satzgrenze

Funktion des Punktes (Auswahl)

- Kennzeichnung von Abkürzungen, wie z.B. (zum Beispiel) oder etc. (et cetera)
- Nummerierungen, beispielsweise Kapitelangaben wie 1. oder bei Aufzählungen
- Ordnungszahlen, wie der 2. Weltkrieg (gesprochen der *Zweite* Weltkrieg)
- Formatierungszeichen, im Deutschen durch einen Punkt gekennzeichnet, zum Beispiel "100.000" Menschen.
- Auslassungspunkte
- Domainspezifische Bezeichner (Auswahl):
  - IPv4 Adressen, wie "31.15.64.162"
  - URI (*Uniform Resource Identifier* , wie "https://blog.fefe.de"
- Datumsangaben, wie zum Beispiel *02. Dezember 2013*

Satz und Satzgrenze

Satzgrenzen (Aus Duden Regeln zur Rechtschreibung)

- *Ganzsätze*: Durch Satzzeichen wie Komma, Semikolon, Doppelpunkt, Gedankenstrich und Klammern gegliedert
- Unterscheidungen beim Doppelpunkt:
  - *Er sagte: "Ich komme morgen."*[1]

  Satzgrenze: Dem Doppelpunkt geht wörtlich wiedergegebene Textstelle voraus

  - *Haus und Hof, Geld und Gut: alles ist verloren.*

  Keine Satzgrenze: Zusammenfassung des vorher Gesagten.
- Semikolon: Keine Satzgrenze, mit dem Semikolon erreicht man zwar eine höhere Abgrenzung als mit dem Komma, aber: im Vergleich mit einem Satzpunkt ist die Abgrenzung wesentlich schwächer.

[1] Aus DUDEN (2006), Seite 71ff

Literatur - Satzendeerkennung

- Palmer und Hearst(Siehe Palmer & Hearst (1997)): *SATZ* System, inkl. *Part-Of-Speech* Tagging, Kontextvektor und Klassifizierung durch Lernalgorithmen
- Andrei Mikheev (1)[1]: Ebenfalls *Part-Of-Speech* inkl. Dokumentzentrierte Heuristiken, Methode zur Abkürzungserkennung wird beschrieben
- Andrei Mikheev (1)[2]: U.a. Auffinden von Eigennamen, Frequenzliste der häufigsten Satzanfänge, Klein- und Großschreibung bestimmter Wörter im Text

[1] Siehe Mikheev (2000)
[2] Siehe Mikheev (2002)

Satz und Satzgrenze

Probleme bei der Satzendeerkennung (Kleine Auswahl)

- Parenthesen[1]:

  Eines Tages – es war mitten im Sommer! – hagelte es.

- Abkürzung am Satzende:

  Pittsburgh wiped up the earth with us. The Reds walked all over us. The Cubs in the last two games have given us the razz.

- Datumsangaben am Satzende:

  Die Vorlesung fand statt am 20.02.2014. Beschrieben war das im LSF.

[1] Übernommen aus DUDEN (2006), Seite 90

EOS_V2 - das Programm zur Satzendeerkennung

Geschichte

- *EOS* ("End of sentence") wurde seit dem Sommersemester 2009 von mehreren Studenten unter Leitung von Herrn Dr. Maximilian Hadersbeck weiterentwickelt:
- Mitwirkende (Sommersemester 2009): Susanne Peters, Michael Mandl, Daniel Bruder und Jonathan Cummings
- Mitwirkende (Sommersemester 2010): Florian Fink (UTF-8 Bibliothek), Susanne Peters, Daniel Bruder, Dino Azzano, Estelle Perez und Simon Thum
- Mitwirkende (Wintersemester 2012/2013): Benno Weck, Jasmin Chebib, Martin Röhrs, Matthias Lindinger, Eamonn Lawlor, Angela Krey und Stefan Schweter

Aufbau - Agenten (1)

- Audhumbla: Markiert potenzielle Satzgrenzen
- Abbreviation: Abkürzungserkennung via Abkürzungsliste, morphologische Abkürzungserkennung bei unbekannten Abkürzung befindet sich in Testphase
- Cross: Überprüft, ob das Wort nach dem potenziellen Satzende im Text nochmals kleingeschrieben, oder in einer eingebundenen Frequenzliste von häufig kleingeschriebenen Wörtern existiert. Trifft eines der beiden Fälle zu, so handelt es sich wahrscheinlich um ein Satzende.

Aufbau - Agenten (2)

- Minuscule: Regelbasierter Agent, der überprüft, wie es nach einem potenziellen Satzende im Text weitergeht (zwei Zeilenumbrüche: Satzende, Kleinbuchstabe: Kein Satzende etc.)
- Primus: Überprüft, ob das Wort nach einem potenziellen Satzende in einer Liste von häufigen Satzanfängen steht.
- Regex: Agent kann anhand einer Liste von regulären Ausdrücken potenzielle Satzenden bewerten. Auszug:

```
-/(str\.)$/i
```

EOS_V2 - das Programm zur Satzendeerkennung

Scoreberechnung

- Jedes potenzielle Satzende wird von den genannten Agenten *bewertet* - jeder Agent liefert dafür einen Score zurück, der im Bereich von **-127** bis **+127** liegt
- **-127** bedeutet: Der Agent vermutet, dass *kein* Satzende vorliegt
- **+127** bedeutet: Der Agent vermutet, dass *ein* Satzende vorliegt
- Nach dem alle Agenten die potenzielle Satzgrenze bewertet haben, werden alle Bewertungen zusammengerechnet. Liegt der Wert über einen bestimmten *Schwellenwert* so liegt ein Satzende vor.
- Gewichtung der Agenten und der *Schwellenwert* können per Konfigurationsdatei gesteuert werden.

Performanzoptimierung

Datenstrukturen für effiziente in-memory-Speicherung

Das Programm arbeitet sehr viel mit (ein bzw. zweispaltigen) Listen. Welche Datenstrukturen sind dafür geeignet? Aus Nikolas Askitis[1] "Efficient Data Structures for Cache Architectures":

- (Einfach) verkettete Listen und Arrays
- Baum-artige Datenstrukturen, wie binäre Suchbäume, AVL-Bäume, Rot-Schwarz-Bäume, Splay Bäume
- Tries (oder: Prefixbäume bzw. "digital trees"[2]), wie Patricia Trie oder Ternary Search Trie (TST)
- Hashes bzw. Hashtabellen
- Automaten

[1] Siehe Askitis (2007)
[2] Siehe Gonnet & Baeza-Yates (1991)

Die *Array Hash Map* als speichereffiziente Datenstruktur für Frequenzlisten

Aus Nikolas Askitis "Efficient Data Structures for Cache Architectures":
Grundlage: *Array Hash* - Kollisionen in dieser Hashtabelle werden nicht mittels verketteter Listen aufgelöst, sondern in einem zusammenhängenden (längen-codierten) Array

- freie Implementation einer *Array Hash* Tabelle[1] wurde verwendet und weiterentwickelt
- *Array Hash Map* enthält zusätzliches Feld für einen Wert (*value*)
- Speicherung von *Key-Value*-Paaren nun performant möglich

[1] Ursprungsversion `https://github.com/chris-vaszauskas/hat-trie` - aktuell nicht mehr vorhanden

Performanzoptimierung

Performanzvergleich: *map* und *unordered_map* gegen *Array Hash Map*

Datenstruktur	Speicherverbauch
std::map	2.83 GB
std::unordered_map	2.66 GB
array_hash_map (Slotgröße: 524288)	589.5 MB
std::vector (Kein Value)	1.57 GB

Testsystem: AMD FX-8150, Kernel: 3.11.6, GCC: 4.8.2
Testdatensatz: 28 Millionen Wörter (*unique*), rund 300 MB groß, Datensatz von Nikolas Askitis Webseite

Performanzoptimierung

Performanzvergleich: *map* und *unordered_map* gegen *Array Hash Map*

Datenstruktur	Laufzeit/ Durchlauf
std::map	1m2.607s
std::unordered_map	0m24.689s
array_hash_map (Slotgröße: 524288)	0m14.563s
std::vector (Kein Value)	0m5.600s

Testsystem: AMD FX-8150, Kernel: 3.11.6, GCC: 4.8.2
Testdatensatz: 28 Millionen Wörter (*unique*), rund 300 MB groß, Datensatz von Nikolas Askitis Webseite

Performanzoptimierung

Performanzvergleich: *map* und *unordered_map* gegen *Array Hash Map*

Einfügen und Suchen:

Datenstruktur	Laufzeit/ Durchlauf
std::map	3m43.235s
std::unordered_map	0m54.369s
array_hash_map (Slotgröße: 524288)	0m32.227s

Testsystem: AMD FX-8150, Kernel: 3.11.6, GCC: 4.8.2
Testdatensatz (Einfügen): 28 Millionen Wörter (*unique*), rund 300 MB groß, Datensatz von Nikolas Askitis Webseite Testdatensatz (Suche): 177 Millionen Wörter, rund 1,2 GB groß, Datensatz von Nikolas Askitis Webseite

Weitere Optimierungen

- *C++11*[1] Features wie: Breitenbegrenzung[2] eines *enums*, *pop_back()*[3] statt *erase()* Methode für das Entfernen des letzten Buchstabens aus einer Zeichenkette
- *strcmp* und *memcpy* Methoden aus Felix von Leitner's[4] *dietlibc*[5] statt aus der *GNU C*[6] Bibliothek (*dietlibc* Methoden sind rund 5 Prozent schneller)

---

[1] Siehe ANSI (2012)
[2] Siehe ANSI (2012), Seite 159 und Grimm (2012), Seite 232
[3] Siehe Stroustrup (2013), Seite 1041
[4] Siehe `https://blog.fefe.de/`
[5] Siehe `https://www.fefe.de/dietlibc/`
[6] Siehe `http://www.gnu.org/software/libc/`

Coding Style

Linux Kernel coding style

- Ein einheitlich definierter Programmierstil (*coding style*) erleichtert die Lesbarkeit, trägt zur Verständlichkeit und zur Wartbarkeit des Quelltextes bei
- *EOS* verwendet den *Linux kernel coding style*[1]

[1] Siehe `https://www.kernel.org/doc/Documentation/CodingStyle`

Coding Style

Weitere *coding styles*

- LLVM: `http://llvm.org/docs/CodingStandards.html`
- Google: `http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml`
- Chromium: `http://www.chromium.org/developers/coding-style`
- Mozilla: `https://developer.mozilla.org/en-US/docs/Developer_Guide/Coding_Style`
- WebKit: `http://www.webkit.org/coding/coding-style.html`

Coding Style

- Ein einheitlicher Programmierstil kann auf mögliche "Fehler" *hinweisen*[1]:

```
1  if ((err = SSLHashSHA1.update(&hashCtx,&clientRandom))!=0)
2          goto fail;
3  if ((err = SSLHashSHA1.update(&hashCtx,&serverRandom))!=0)
4          goto fail;
5  if ((err = SSLHashSHA1.update(&hashCtx,&signedParams))!=0)
6          goto fail;
7          goto fail;
8  if ((err = SSLHashSHA1.final(&hashCtx, &hashOut))!=0)
9          goto fail;
```

[1] Siehe `http://opensource.apple.com/source/Security/Security-55471/libsecurity_ssl/lib/sslKeyExchange.c`

Modultests bzw. *Unittests*

Unittests

- Für jeden *EOS* Agenten wurden eigene *Unittests* implementiert
- *Unittests* ermöglichen u.a. schnelle und wiederholbare Tests
- Aber: *Unittests* alleine garantieren noch nicht die Fehlerfreiheit des Programmes
- Als *Unittest* Framework wird die *Boost* Testbibliothek[1] verwendet

[1] Siehe `http://www.boost.org/doc/libs/1_55_0/libs/test/doc/html/index.html`

Unittests - Motivation (Auswahl)

- *Boundary checks*[1] als Testfälle können "hilfreich" sein[2]:

```
A missing bounds check in the handling of the
TLS heartbeat extension can be used to reveal
up to 64k of memory to a connected client or
server.
```

```
1 /* Read type and payload length first */
2 if (1 + 2 + payload + 16 > s->s3->rrec.length)
3     return 0; /* silently discard per RFC 6520 sec. 4 */
```

[1] Siehe http://git.openssl.org/gitweb/?p=openssl.git;a=commitdiff;h=96db902

[2] Mehr Informationen zum OpenSSL *heartbleed* Bug unter http://heartbleed.com/

Modultests bzw. *Unittests*

Unittests - Motivation (Auswahl)

- Bei Änderungen am Quelltext kann erneut auf Richtigkeit "getestet" werden[1]:

```
1 /* Don't add uninitialised data.
2     MD_Update(&m,buf,j);
3 */
```

- Wenn nicht richtig getestet wird, kann das u.a. diese Konsequenzen haben:

Luciano Bello discovered that the random number generator in Debian's openssl package is predictable.

[1] Siehe Zufallszahlengenerator Bug in Debian https://www.schneier.com/blog/archives/2008/05/random_number_b.html

Evaluation

Brown Korpus

Das Brown Korpus wurde aus *NLTK*[1] extrahiert und bereinigt. Evaluiert wurden die ersten 1200 Zeilen:

$$Precision = \frac{tp}{tp + fp} = \frac{10.146}{10.146 + 17} \approx 99,83\%$$

$$Recall = \frac{tp}{tp + fn} = \frac{10.146}{10.146 + 9} \approx 99,91\%$$

[1] Siehe `http://nltk.org/`

Evaluation

Europarl Korpus

Der englische Europarl Korpus wurde ebenfalls mittels Python Skript aus NLTK extrahiert, und wie das Brown Korpus entsprechend bereinigt. Im Gegensatz zum Brown Korpus wurde das Europarl Korpus komplett evaluiert:

$$Precision = \frac{tp}{tp + fp} = \frac{18.738}{18.738 + 5} \approx 99,97\%$$

$$Recall = \frac{tp}{tp + fn} = \frac{18.738}{18.738 + 11} \approx 99,94\%$$

Evaluation

Penn Treebank Korpus

$$Precision = \frac{tp}{tp + fp} = \frac{10.716}{10.716 + 111} \approx 98,97\%$$

$$Recall = \frac{tp}{tp + fn} = \frac{10.716}{10.716 + 73} \approx 99,32\%$$

Evaluation

Dostojewski: "Verbrechen und Strafe" (1924)

Der Text "Verbrechen und Strafe" von Fjodor Dostojewski stammt aus dem Jahre 1924, und wurde von `http://www.zeno.org` zur Evaluation verwendet:

$$Precision = \frac{tp}{tp + fp} = \frac{11.797}{11.797 + 6} \approx 99,95\%$$

$$Recall = \frac{tp}{tp + fn} = \frac{11.797}{11.797 + 48} \approx 99,59\%$$

Evaluation

Dostojewski: "Schuld und Sühne" (1956)

Der Text "Schuld und Sühne" von Fjodor Dostojewski stammt aus dem Jahre 1956, und wurde von `http://zeno.org` zur Evaluation verwendet:

$$Precision = \frac{tp}{tp + fp} = \frac{11.348}{11.348 + 3} \approx 99,97\%$$

$$Recall = \frac{tp}{tp + fn} = \frac{11.348}{11.348 + 28} \approx 99,75\%$$

Fazit und Ausblick

Zusammenfassung

- Das Programm liefert sehr gute Ergebnisse für die getesteten Sprachen Deutsch und Englisch
- *Array Hash Map* eignet sich hervorragend für diverse Frequenzlisten, die im Programm verwendet werden
- *Array Hash Map* ist *frei* verfügbar (*GNU General Public License* 2) unter `http://schweter.eu/subs/ahm/index.html`
- *EOS* soll demnächst ebenfalls als *freie* Version veröffentlicht werden
- Demoseite verfügbar unter `http://demomax.cis.uni-muenchen.de/home_demos/eosv3/index.html`

Quellenangaben I

ANSI (2012): ISO/IEC 14882-2011.

ASKITIS, Nikolas (2007): Efficient Data Structures for Cache Architectures.

DUDEN (2006): Deutsche Rechtschreibung, Regeln und Wörterverzeichnis.

DÜRR, Michael (2006): *Deskriptive Linguistik*. https://encrypted.google.com/books?id=VDCA5j2Kc0gC (Abgerufen 23.11.2013): Vandenhoeck und Ruprecht

GONNET, G.H., BAEZA-YATES, R. (1991): *Handbook of Algorithms and Data Structures*. (2. Aufl.). Addison-Wesley

GRIMM, Rainer (2012): *C++11*. Addison-Wesley

MIKHEEV, Andrei (2000): Tagging Sentence Boundaries.

MIKHEEV, Andrei (2002): Periods, Capitalized Words, etc.

Quellenangaben II

PALMER, David D., HEARST, Marti A. (1997): Adaptive Multilingual Sentence Boundary Disambiguation.

STROUSTRUP, Bjarne (2013): *The C++ Programming Language*. (4. Aufl.). Addison-Wesley