Programmierkurs Python II – SS 10

Programmierprojekt

Ziel des Projektes ist die Implementierung eines einfachen statistischen Parsers. Das Projekt besteht aus drei Hauptteilen:

1. Implementieren eines einfachen *PCFG-Parsers*
2. Die *Trainings-Infrastruktur*: Die PCFG – Regeln und Regelwahrscheinlichkeiten – werden aus einer Baumbank abgelesen
3. *Evaluation* des Parsers anhand von Korpus-Daten

Hinweis: Da die Wahrscheinlichkeiten sehr schnell sehr klein werden können, bietet es sich an, generell mit „logarithmischen“ Wahrscheinlichkeiten zu rechnen:

- $log(p\,q) = log(p) + log(q)$
- $p < q \Rightarrow log(p) < log(q)$

1 Parser

Implementiere den CYK-Algorithmus für probabilistische kontextfreie Grammatiken. Der Parser soll statt einer Liste von Wörtern eine Liste von Paaren aus Wörtern und ihrer Wortart (POS) als Eingabe verwenden, d.h. die Wörter der Eingabekette sind bereits mit ihren POS-Tags versehen. Der Algorithmus soll die POS als Terminalsymbole auffassen (siehe Teil 2). Die eigentlichen Wörter sollen nur bei der Extraktion des Ableitungsbaums aus der Chart als Blätter an den Baum angehängt werden.

Für den Parser soll der ursprüngliche CYK-Algorithmus aus der Vorlesung etwas erweitert werden:

- Der eigentliche CYK-Algorithmus funktioniert nur mit Grammatiken in Chomsky-Normal-Form, siehe Vorlesung. Erweitere den Algorithmus so, dass er auch mit unären Regeln arbeiten kann.
- Erweitere die Implementierung außerdem so, dass die Binarisierung wieder rückgängig gemacht werden kann.
- Zur bequemeren Evaluierung (vgl. Aufgabe 3.) soll die Implementierung so angepasst werden, dass sie alternativ Ableitungsbäume als Eingabe akzeptiert, aus denen die Präterminalsymbole extrahiert und als Eingabe für das Parsen verwendet werden. Aus dem Baum wird also der eigentliche Satz und die POS-Tags „ausgeschnitten“. Das Baumformat ist im nächsten Abschnitt erläutert.

Bonusaufgabe: Erweitere die Implementierung so, dass sie mit einem Tagger kombiniert werden kann, so dass der Parser auch mit „freiem Text“ funktioniert. Hier bieten sich z.B. die Tagger aus dem NLTK an. Beachte, dass die von diesen Taggern verwendeten POS-Symbole leicht von den POS-Symbolen in der Penn-Treebank leicht abweichen können.[1]

2 Trainings-Infrastruktur

Die Penn Treebank ist ein Korpus, das u.A. manuell annotierte syntaktische Ableitungsbäume für das Wall-Street Journal (WSJ) enthält. Extrahiere aus den WSJ-Abschnitten der Baumbank eine PCFG in Chomsky-Normalform (mit unären Regeln). Die Originaldaten liegen auf den CoLi-Servern unter
`/proj/corpora/penntreebank3/parsed/mrg/wsj`.

Wir werden eine leicht aufbereitete Variante der Baumbank zur Verfügung stellen, bei der genau ein (vollständiger) Baum auf einer Zeile steht.

Bäume sind in Listen-Notation kodiert: (Nichtterminal Baum Baum ...).

Manche Nichtterminale (z.B. `NP-SBJ` für Subjekte) sind mit einem zusätzlichen „Tag“ (hier: `SBJ`) annotiert. Diese Tags sollten bei der Extraktion der PCFG ignoriert werden. Ausserdem ist zu beachten, dass die Bäume „leere“ Elemente enthalten können (Kategorie: `-NONE-`). Diese müssen vor der Extraktion der PCFG entfernt werden. Beachte, dass man ggf. auch den Elternknoten entfernen muss, wenn das leere Element das einzige Kind des Knotens ist.

Beachte, dass beim Extrahieren der Grammatik aus der Baumbank die eigentlichen Terminalsymbole (Wörter) ignoriert werden müssen – stattdessen sollen deren Elternknoten (die *Parts of Speech*, POS) der Terminalsymbole als Terminalsymbole aufgefasst werden.

Gehe ansonsten für die Induktion von Grammatik und Wahrscheinlichkeiten so vor wie in der Vorlesung besprochen.

3 Evaluation

Implementiere zwei Funktionen, um *Labelled Precision* und *Labelled Recall* zu berechnen. In den Formeln unten steht $\text{Baum}_{\text{Referenz}}$ für den Baun aus der Baumbank und $\text{Baum}_{\text{Parser}}$ für den von Deinem Parser berechneten Baum. Grundsätzlich ist immer die Anzahl der Konstituenten gemeint.

$$\text{Labelled Precision} = \frac{\text{korrekte Konstituenten im Baum}_{\text{Parser}}}{\text{Konstituenten Baum}_{\text{Parser}}}$$

$$\text{Labelled Recall} = \frac{\text{korrekte Konstituenten Baum}_{\text{Parser}}}{\text{Konstituenten Baum}_{\text{Referenz}}}$$

[1] Anleitung für POS-Tagger mit NLTK:
`http://nltk.googlecode.com/svn/trunk/doc/howto/tag.html`

Die Konstituenten sind Teilbäume des Ableitungsbaums; es wird also für alle Teilbäume des Baums getestet, ob sie in der Referenz vorkommen. Eine Konstituente gilt als korrekt, wenn sie die gleiche Teilkette der Eingabe überspannt und mit dem gleichen Nichtterminalsymbol etikettiert ist wie eine Konstituente des Baums aus der Baumbank. „Leere“ Konstituenten (-NONE-) sowie Satzzeichen sollen bei dem Vergleich ignoriert werden.

Du kannst deine Implementierung testen, indem du die Ausgabe mit der entsprecheden Ausgabe von EVALB vergleichst[2].

Erweitere die Parser-Implementierung so, dass sie – wenn ein Baum als Eingabe angegeben wird (sieh Teil 1) – die vom Parser berechneten Ableitungsbäume direkt mit der Referenz-Eingabe bzgl. LP/LR vergleicht.

Extrahiere die Grammatik für Abschnitte 0-22 des WSJ, und teste die Grammatik für die Bäume in Abschnitt 23. Da die Grammatik sehr groß werden kann, solltet Ihr zum *Entwickeln* des Codes nur einen Teil der Baumbank betrachten. Falls die Laufzeit der *finalen* Evaluation zu lange dauert (max. 1h auf den CoLi-Clustern), reicht es, Sätze mit maximal 15 Wörtern zu betrachten.

4 Dokumentation

Schreibe eine kurze ($\frac{1}{2}$ - 1 Seite) Dokumentation, die klar sagt, wie das Programm benutzt werden muss. Gegeben die Trainingsdaten in einem beliebigen Ordner, müssen die Befehle zum Trainieren des Parsers, zum Parsen des Testkorpus und zum Evaluieren aufgeführt werden. (Falls Teile davon kombiniert sind zu einem Befehl, sollte das so da stehen.) Bitte gebt auch eine Zeitabschätzung, wie lange Euer Programm für das Trainieren / Parsen der angegebenen Trainingsdaten braucht und welche Einschränkungen (auf Wortlänge 15 z.B.) ihr ggf. gemacht habt.

Abgabe bis Montag, 13.09.10

[2] `http://nlp.cs.nyu.edu/evalb/`