Projekt 2: Ein einfacher CYK-Parser für probabilistische kfG

Im zweiten Projekt wollen wir einen einfachen Parser für probabilistische kontextfreie Grammatiken (PCFG) implementieren. Ausgangspunkt ist der PCYK-Parser aus dem 8. Übungsblatt.

Das Projekt gliedert sich in folgende Teile:

1. Implementieren Sie ein Programm, das eine Baumbank einliest, aus den Bäumen eine PCFG extrahiert und diese in einem geeigneten Format speichert.
2. Erweitern Sie die zur Verfügung gestellte Beispielimplementierung des PCYK-Parsers um Funktionen, die die Grammatik einlesen und die Regeln in Chomsky-Normalform überführen.
3. Sie werden feststellen, dass die Beispiel-Implementierung des Parsers sehr langsam ist. Verbessern Sie die Laufzeit: Mit zwei relativ einfachen Änderungen lässt sich die Effizienz des Parsers spürbar steigern.
4. Evaluieren Sie den Parser: Implementieren Sie ein Programm, das *Labelled Precision* und *Labelled Recall* der Parse-Bäume berechnet.
5. Schreiben Sie eine kurze Dokumentation, in der Sie Ihre Implementierung kurz beschreiben und die Evaluationsergebnisse zusammenfassen. Umfang: c.a. 2 Seiten.

Teil 1: Extraktion einer PCFG aus einer Baumbank.

Die zur Verfügung gestellte Datei „wsj-train.mrg“ enthält Bäume aus dem Wall-Street-Journal-Teilkorpus der Penn Treebank in einem leicht vereinfachtem Format. Aus diesen Bäumen kann man leicht eine kontextfreie Grammatik extrahieren: Jeder „Teilbaum“ (genauer: Baumfragment) der Tiefe 1 entspricht einer Regel der Grammatik. Zum Beispiel kann man aus

(S (NP (Det die) (N Studentin)) (VP (V arbeitet))

die Regeln

S → NP VP NP → Det N VP → V Det → die N → Studentin V → arbeitet

extrahieren. Um eine PCFG abzulesen, muss zusätzlich gezählt werden, wie oft die einzelnen Regeln vorkommen. Die Wahrscheinlichkeit einer Regel kann dann wie folgt abgeschätzt werden:

$$P(X \rightarrow Y_1 \dots Y_k) = \text{Count}(X \rightarrow Y_1 \dots Y_k) \,/\, \sum_{\alpha} \text{Count}(X \rightarrow \alpha)$$

Implementieren Sie ein Programm „train.py“, das die Bäume aus der Datei „wsj-train.mrg“ einliest, eine PCFG extrahiert und in einem geeigneten Format speichert. Es empfiehlt es sich, die lexikalischen Regeln (mit genau einem Terminalsymbol auf der rechten Seite) und die nicht-lexikalischen Regeln in separaten Deteien („lexicon.txt“ und „rules.txt“) zu speichern.

Wie bei dem ersten Teilprojekt stellt sich auch hier das Problem, dass bei der Anwendung des Parsers auf neue Daten unweigerlich „unbekannte“ Wörter vorkommen werden. Wir wollen hier den gleichen Lösungsansatz wie im ersten Teilprojekt verfolgen: Ersetzen Sie alle Wörter in der Baumbank, die genau einmal vorkommen, durch das Pseudowort „OOV“, bevor Sie die PCFG aus den Bäumen ablesen. (Statt Wörter zu ersetzen können Sie alternativ auch einfach beim Zählen der Regelhäufigkeiten „OOV“ anstelle des eigentlichen Wortes zählen.)

Teil 2: Der CYK-Parser.

Die Datei „pcyk.py“ enthält eine Beispiel-Implementierung des CYK-Parsers für PCFGen. Erweitern Sie die Implementierung um Funktionen, die die PCFG aus Teil 1 einlesen und in Chomsky-Normalform (CNF) überführen. Wenn Sie den ersten Teil korrekt implementiert haben, dann enthält die Grammatik keine Kettenregeln (mit genau einem Nichtterminal auf der rechten Seite). Um die extrahierte Grammatik in CNF zu überführen genügt es deshalb, Regeln

$X \rightarrow Y_1 \ldots Y_{n-1}\ Y_n$ [p]

mit mehr als 2 Nichtterminalsymbolen auf der rechten Seite durch die Regeln

$X \rightarrow N\ Y_n$ [p]

$N \rightarrow Y_1 \ldots Y_{n-1}$ [1.0]

zu ersetzen, wobei N ein neues (in den übrigen Regeln der Grammatik nicht vorkommendes) Nichtterminalsymbol ist und *p* die Regelwahrscheinlichkeit angibt. Die Regel $N \rightarrow Y_1 \ldots Y_{n-1}$ muss ggf. weiter binarisiert werden (wenn $n > 3$).

(Das oben beschriebene Verfahren nennt sich „Linksbinarisierung“. Alternativ hätten wir auch „nach rechts“ binarisieren und die Regel oben durch $X \rightarrow Y_1\ N$ [p], $N \rightarrow Y_2 \ldots Y_n$ [1.0] ersetzen können.)

(Alternativ können Sie selbstverständlich die Binarisierung bereits in Teil 1 vornehmen.)

Hinweis: Bei der Implementierung bietet es sich an, neue Nichtterminalsymbole N durch Konkatenation der entsprechenden rechten Regelseiten zu erzeugen. Zum Beispiel kann man die Regel

("VP", ["V", "NP", "PP"])

wie folgt binarisieren:

("VP", ["V+NP", "PP"])

("V+NP", ["V", "NP"])

Dies hat den Vorteil, dass man den Nichtterminalsymbolen direkt ansieht, ob sie aus der Baumbank stammen oder durch Binarisierung entstanden sind. (Das ist wichtig bei der Evaluation.)

Teil 3: Laufzeit verbessern

Sie können jetzt den Parser auf die Testdaten („wsj-test.txt“) anwenden. Sie werden feststellen, dass der Parser sehr langsam ist. Ein Grund dafür liegt in der Struktur der Implementierung des Algorithmus: In der innersten for-Schleife versucht der Parser, benachbarte Teilkonstituenten zu größeren Konstitutenten zu kombinieren. Dazu iteriert die Implementierung erst über alle Regeln $L \rightarrow R_1\ R_2$ und prüft danach, ob R_1 in T[j, k] und R_2 in T[k, i] enthalten ist. Sie werden feststellen, dass es deutlich effizienter ist, wenn man erst über alle Elemente R_1 in T[j,k] und alle Elemente R_2 in T[k,i] iteriert und erst dann prüft, ob es eine passende Regel der Form $L \rightarrow R_1\ R_2$ gibt. (Warum ist das so?) Implementieren Sie diesen Ansatz. Dazu muss die Datenstruktur für die Regeln der Grammatik geändert werden. Statt einer Liste von Tupeln bietet es sich an, ein (ggf. geschachteltes) Wörterbuch (dict) zu verwenden, dessen Schlüssel die Nichtterminale auf der rechten Regelseite sind.

Eine zweite Optimierung besteht darin, in jeder Zelle T[i,j] jeweils nur die *n* wahrscheinlichsten Teilkonstituenten zu speichern („Beam Search“). Durch diese Änderung verschlechtert sich potentiell die

Performanz des Parsers (Warum?), aber für Werte $n \geq 50$ ändert sind die Perfomanzeinbußen marginal.

Hinweis: Während der Entwicklung bietet es sich an, den Parser nur auf kurzen Sätzen zu testen. Die Beispiel-Implementierung sieht vor, dass man mit der Kommandozeilenoption --maxlen n die maximale Satzlänge festlegen kann. (Längere Sätze werden ignoriert.) Mit der Kommandozeilenoption --beam n kann angeben, wie viele (Teil-)Konstituenten jeweils in den Zellen der Chart gespeichert werden sollen.

Teil 4: Evaluation des Parsers

Schließlich wollen wir wissen, wie gut unser Parser funktioniert. Implementieren Sie dazu ein Programm „eval.py“, das *Labelled Precision (LP)* und *Labelled Recall (LR)* der Parse-Bäume berechnet:

LP = # korrekte Konstituenten / # Konstituenten im Parse-Baum

LR = # korrekte Konstituenten / # Konstituenten im Referenz-Baum

wobei eine Konstituente (Knoten) im Parse-Baum als korrekt gilt, wenn es einen Knoten im Referenz-Baum gibt, der die gleiche Eingabe überspannt und die beiden Knoten mit dem gleichen Nichtterminalsymbol etikettiert sind. Die Referenz-Bäume finden sich in der Datei „wsj-test.mrg“.

Zu beachten: Knoten, die mit einem durch Binarisierung entstandenen Nichterminalsymbol („X+Y“) etikettiert sind, sollen bei der Evaluation vollständig ignoriert werden. D.h. die Berechnung von LP und LR für den Baum

(S (NP …) (VP (VP+NP (VP …) (NP …)) (PP …))

soll von 6 Knoten bzw. Konstituenten ausgehen, nicht 7, da „VP+NP“ durch die Binarisierung der Grammatikregeln entstanden ist. Wenn

(S (NP …) (VP (VP …) (NP …) (PP …))

der Referenzbaum aus den Testkorpus ist, dann ist LP und LR in diesem Fall 1!

Teil 5: Dokumentation

Schreiben Sie eine kurze Dokumentation, in der Sie Ihre Implementierung kurz beschreiben und die wesentlichen Evaluationsergebnisse zusammenfassen: Geben Sie die durchschnittlichen LP- und LR-Werte sowie die Laufzeit, die maximale Länge der Sätze, die Sie in der Evaluation betrachtet haben (--maxlen), sowie die maximale Anzahl an Konstituenten in den Zellen der Chart (--beam).

Hinweis: Die Referenzimplementierung benötigt auf einem Mittelklasse-Notebook rund 10 Minuten, um alle (1141) Sätze mit maximal 25 Wörtern aus der Baumbank zu parsen (--beam 50). Die LP ist 0,89, LR ist 0,88. Wenn man --beam auf den Wert 20 setzt, veringert sich die Laufzeit auf weniger als 3 Minuten; LP und LR verringern sich marginal.

Beachten Sie: In der Literatur werden deutlich geringere Werte für LP und LR angegeben. Wir verwenden ein vereinfachtes Test-Set, deshalb sind die Werte für LP und LR nicht mit den in der Literatur angegebenen Werten vergleichbar!

Einreichfrist: 1. April 2022