Annotation als Suche

Rebekka Hubert Michael Staniek Simon Will

November 29, 2016

Übersicht

Motivation

- Menschliche Annotation
  - dauert und kostet
  - bei Dependenzparses immer noch ungeschlagen.
- Wir suchen eine Möglichkeit, Annotatoren zu unterstützen
  - Basierend auf vielen möglichen Parses für einen Satz Fragen an den Annotator stellen
  - anhand der Fragen zum optimalen Parsebaum gelangen

Für wen?

- Das Programm soll für Annotatoren gedacht sein.
  - Diese Userklasse hat nicht unbedingt viel Programmiererfahrung.
- Eine gewisse Grunderfahrung im Annotieren wird vorausgesetzt.

Requirements

Um die Anpassungsfähigkeit des Systems nicht von Anfang an einzuschränken, bietet es sich an, das System in drei Module aufzuteilen:

- Preprocessing:
  - Generierung der möglichen Parsebäume unter Betrachtung und Abänderung verschiedener Parser
  - Übergabe der k-besten Parsebäume an das System zur Fragegenerierung
- Fragengenerierung
  - System zur Fragegenerierung mithilfe eines Algorithmus (oder mehreren) erhält über Preprocessing-Schnittstelle die Parsebäume

Requirements

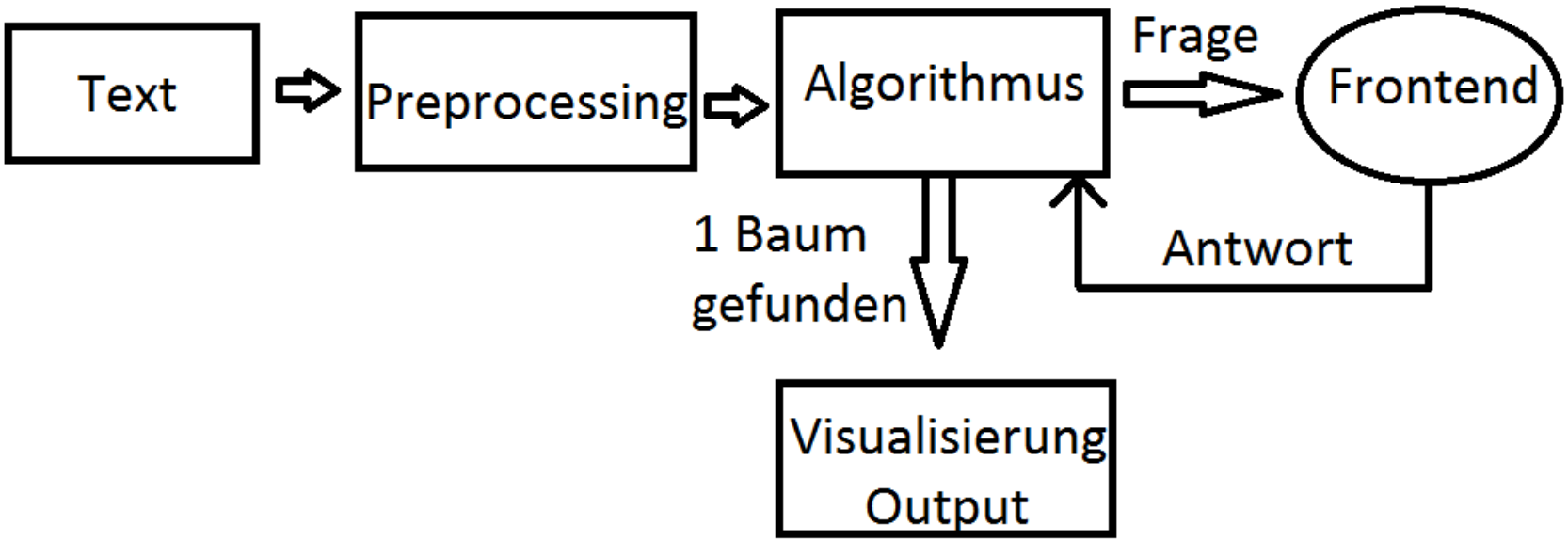
- Benutzerschnittstelle:
  - Übermittlung der Fragen an den Annotator
  - Übergabe der Antwort des Annotators an den Algorithmus
  - *Darstellung* des Parsebaums ist unabhängig zu diesem Prozess

Requirements

Die Schnittstellen zwischen diesen Modulen sind wie folgt:

- Preprocessing $\rightarrow$ Fragengenerierung:
- Fragengenerierung (Server) $\leftrightarrow$ Benutzerschnittstelle (Client)

Übersicht


Text
Preprocessing
Algorithmus
Frage
Frontend
1 Baum gefunden
Antwort
Visualisierung Output

Algorithmus

- Generiere aus Parseforest die Frage, welche bei ihrer Beantwortung die meisten Bäume rausfiltert.
  - Suche des Tupels, das den Suchraum am ehesten halbiert:

a $\leftarrow$ len(parses) ▷ Anzahl der Parse-Bäume.
for each tuple **do**
    score $\leftarrow \mathrm{abs}(\mathrm{count}(\mathrm{tuple}) - \frac{a}{2})$
end for

- Nimm den Tupel als Frage

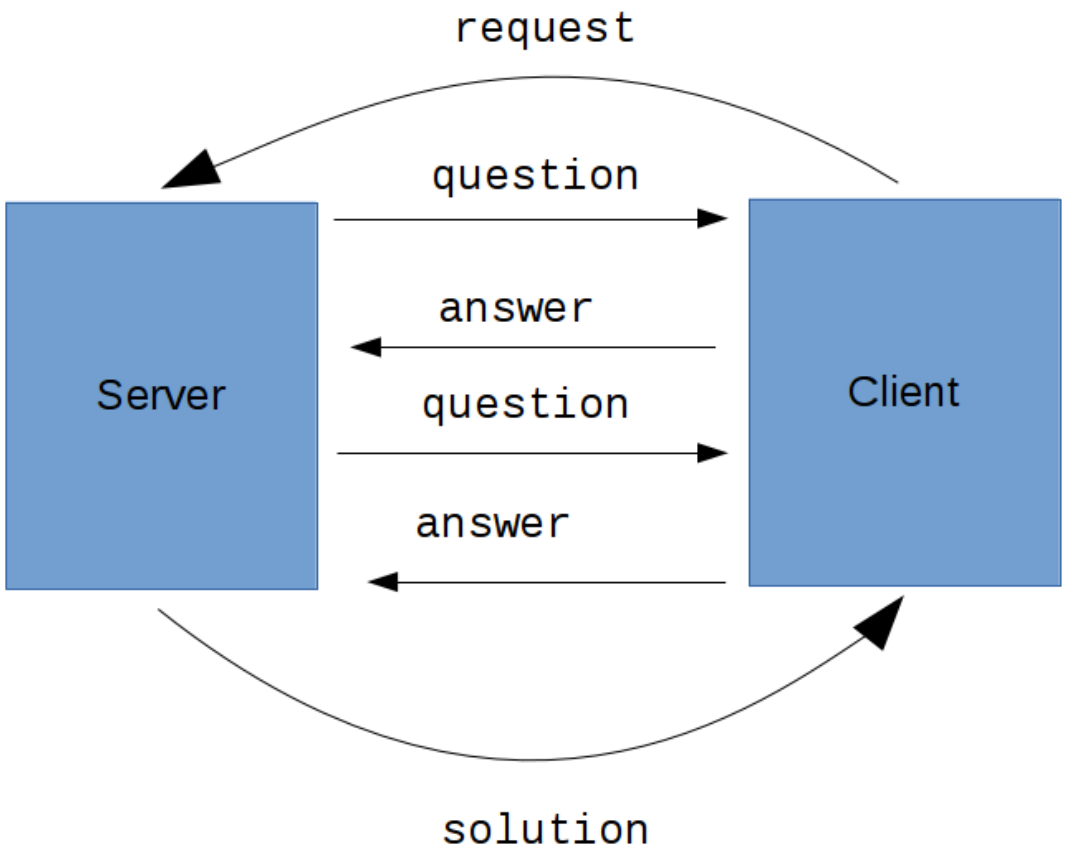
Abhängigkeiten

- Python 3.4 order neuer (aufgrund des Pakets `asyncio`)
- Parser, der k-best Parses liefert
- TCP- oder UNIX-Sockets
- JSON

Client-Server-Kommunikation I

- Client und Server kommunizieren streambasiert über eine offen gehaltene Verbindung.
- Das geschieht entweder über ein TCP- oder ein UNIX-Socket.
- Jede Nachricht besteht aus genau einem JSON-Objekt, das den Schlüssel `"type"` enthalten muss.
- Es gibt die fünf Typen `request`, `question`, `answer`, `solution` und `error`.

Client-Server-Kommunikation II



Nachricht des Typs `request`

- Ein `request` wird vom *Client* gestellt, um die Annotation eines Satzes einzuleiten.
- Der Client spezifiziert ein Feld außer dem `type`:

  `use_forest` wenn der Server einen schon in einen Wald geparsten Satz verwenden soll. Der Wert des Feldes soll dann eine CoNLL-Repräsentation des Waldes sein.

  `parse_sentence` wenn der Server den angegebenen Satz selbst parsen soll. Der Wert des Feldes soll dann ein ganz normaler Satz sein.

Beispiel: request mit parse_sentence

```
{
    "type": "request",
    "parse_sentence": "Mit Bedacht badet heute ein
       Lurch in einem See."
}
```

Beispiel: request mit use_forest


```
{
  "type": "request",
  "use_forest": "1\tMit\tmit\t_\tADP\tAPPR\t_\t_\t3\t_
     \tMO\t_\t_\n2\tBedacht\tbedacht\t_\tNOUN\tNN\t_\
     t_\t1\t_\tNK\t_\t_\n3\tbadet\tbaden\t_\tVERB\
     tVVFIN\t_\t_\t0\t_\t--\t_\t_\n4\theute\theute\t_
     \tADV\tADV\t_\t_\t3\t_\tMO\t_\t_\n5\tein\tein\t_
     \tDET\tART\t_\t_\t6\t_\tNK\t_\t_\n6\tLurch\
     tlurch\t_\tNOUN\tNN\t_\t_\t3\t_\tSB\t_\t_\n7\tin
     \tin\t_\tADP\tAPPR\t_\t_\t3\t_\tMO\t_\t_\n8\
     teinem\tin\t_\tART\tDET\t_\t_\t9\t_\tNK\t_\t_\n9
     \tSee\tsee\t_\tNOUN\tNN\t_\t_\t7\t_\tNK\t_\t_\
     n10\t.\t--\t_\t.\t$.\t_\t_\t3\t_\t--\t_\t_\n\n1\
     tMit\tmit\t_\tADP\tAPPR\t_\t_\t3\t_\tMO\t_\t_\n2
     \tBedacht\tbedacht\t_\tNOUN\tNN\t_\t_\t1\t_\tNK\
     t_\t_\n3\tbadet\tbaden\t_\tVERB\tVVFIN\t_\t_\t0\
     t_\t--\t_\t_\n4\theute\theute\t_\tADV\tADV\t_\t_
     \t6\t_\tMO\t_\t_\n5\tein\tein\t_\tDET\tART\t_\t_
     \t6\t_\tNK\t_\t_\n6\tLurch\tlurch\t_\tNOUN\tNN\
     t_\t_\t3\t_\tSB\t_\t_\n7\tin\tin\t_\tADP\tAPPR\
```

Nachricht des Typs `question`

- Hat der Server eine Frage gefunden, die er dem Client stellen will, sendet er eine Nachricht des Typs `question`.
- Der Server spezifiziert außer dem `type` die folgenden Felder:

`question` Eine Repräsentation der Frage Der Wert des Feldes steht noch nicht völlig fest, ist er wohl ein weiteres JSON-Objekt.

`remaining_trees` Ein Integer, der angibt, wie viele Bäume noch im Wald enthalten sind.

`sentence` Ein String, der den behandelten Satz enthält.

`fixed_edges` Ein Array von Edge-Objekten, die die Kanten repräsentieren, die in jedem Baum im Wald vorkommen und daher schon feststehen.

Beispiel: question


```
{
  "type": "question",
  "question": {
    "head" : "badet-3",
    "dependent": "Lurch-6",
    "relation": "SB"
  },
  "remaining_trees": 4,
  "sentence": "Mit Bedacht badet heute ein Lurch in
      einem See.",
  "fixed_edges":
    [{
    "head" : "badet-3",
    "dependent": "Mit-1",
    "relation": "MO"
  }, {
    "head" : "Mit-1",
    "dependent": "Bedacht-2",
    "relation": "NK"
  }]
```

Nachricht des Typs `answer`

- Hat der Client auf die Frage des Servers eine Antwort gefunden, sendet er eine Nachricht des Typs `answer`.
- Der Client spezifiziert außer dem `type` die folgenden Felder:

  `question` Ein Edge-Objekt, das der Client bestätigen oder zurückweisen soll.

  `answer` Ein boolescher Wert, der die Frage beantwortet, d. h. die vom Server genannte Kante bestätigt oder zurückweist.

Beispiel: answer

```
{
  "type": "answer",
  "question": {
    "head" : "badet-3",
    "dependent": "Lurch-6",
    "relation": "SB"
  },
  "answer": true
}
```

Nachricht des Typs `solution`

- Hat sich der Wald so weit gelichtet, dass nur noch ein Baum übrig ist, sendet der Server keine weitere Frage, sondern eine Nachricht des Typs `solution`.
- Der Client spezifiziert außer dem `type` das folgende Feld:

  `tree` Eine Repräsentation des übrig gebliebenen Baumes. Der Wert des Feldes steht noch nicht völlig fest, ist er wohl ein weiteres JSON-Objekt.

Beispiel: solution


```
{
  "type": "solution",
  "tree": [
    ["1", "Mit", "mit", "_", "ADP", "APPR", "_", "_",
      "3", "_", "MO", "_", "_"],
    ["2", "Bedacht", "bedacht", "_", "NOUN", "NN", "_"
      , "_", "1", "_", "NK", "_", "_"],
    ["3", "badet", "baden", "_", "VERB", "VVFIN", "_",
      "_", "0", "_", "--", "_", "_"],
    ["4", "heute", "heute", "_", "ADV", "ADV", "_", "_
      ", "3", "_", "MO", "_", "_"],
    ["5", "ein", "ein", "_", "DET", "ART", "_", "_", "
      6", "_", "NK", "_", "_"],
    ["6", "Lurch", "lurch", "_", "NOUN", "NN", "_", "_
      ", "3", "_", "SB", "_", "_"],
    ["7", "in", "in", "_", "ADP", "APPR", "_", "_", "3
      ", "_", "MO", "_", "_"],
    ["8", "einem", "in", "_", "ART", "DET", "_", "_",
      "9", "_", "NK", "_", "_"],
    ["9", "See", "see", "_", "NOUN", "NN", "_", "_", "
```

Nachricht des Typs `error`

- Empfängt der Server vom Client eine unerwartete Nachricht, so antwortet er mit einer Nachricht vom Typ `error`.
- Der Client spezifiziert außer dem `type` folgende Felder:

  `error_message` Ein String, der den Fehler beschreibt.

  `recommendation` Ein String, der eine Empfehlung an den Client ausspricht, wie er den Fehler behandeln soll. Mögliche Werte werden vermutlich `"retry"` und `"abort"` sein.

Beispiel: error

```
{
  "type": "error",
  "error_message": "Received an answer, but no forest
      exists.",
  "recommendation": "abort"
}
```

Features – Pflicht

- Stellen von Fragen der Form: „Ist Token1 Relation von Token2?“ an den Annotator
- Antwort reduziert Anzahl der möglichen Parsebäume bis nur noch einer übrig ist
- fertiger Parsebaum wird dem Nutzer gezeigt
- Nutzer kann Parsebaum abspeichern

Features – Optional

Die Erstellung eines interaktiven GUIs, welches den bisherigen Parsebaum anzeigt und weitere benutzerfreundliche Features anbietet:

- Eine *Undo*-Taste um Fehler zu korrigieren.
- Ein visueller Hinweis darauf, welche Felder des Parsebaums bereits feststehen. Der Annotator erkennt somit frühzeitig, falls eine bestimmte Annotation vom System garnicht betrachtet wird.

Features – Optional

- Implementierung weiterer Algorithmen zur Fragengenerierung.
- Eine breitere Formatsunterstützung bei der Ausgabe der fertigen Parsebäume.

Evaluation

- Automatische Evaluation
  - Methode:
    - der Goldparse wird als Annotator benutzt
    - die Fragen werden an den Goldpars gestellt
    - überprüfen: Entspricht Ergebnis dem Goldpars?
  - Vorteile
    - Überprüfen mehrerer Ergebnisse in derselben Zeit
    - Vermeiden menschlicher Fehler

Evaluationsmaß

- mehrere Evaluationsmaße
  - Minimum Edit Distance
    - Anzahl der nicht übereinstimmenden, gelabelten Kanten
  - Labeled Attachment Score
    - Anzahl der gelabeled Kanten, die mit dem Goldstandard übereinstimmen
  - Error Counter
    - Anzahl der nicht gefundenen Bäume