日本語パイプライン処理のための簡易フレームワークの提案

能地 宏 † 榊原 隆文 * 宮尾 祐介 †

† 総合研究大学院大学 情報学専攻 / 国立情報学研究所
* 東京工業大学大学院 物理情報システム専攻
{noji,yusuke}@nii.ac.jp tsakaki@lr.pi.titech.ac.jp

1 はじめに

本稿では、JVM 上で動作し、また容易に拡張を行うことのできる、日本語処理のためのパインプラインフレームワーク Jigg の紹介を行う。これまで日本語処理のための数多くのソフトウェアが公開されてきたが、一般にそれらは異なる入力及び出力形式を持つため、ユーザーはしばしば利用法の習得、またその出力から欲しい情報を抽出するために多くの時間を費やしてしまう。これは他の言語に対する言語処理でも同じであるが、英語では Stanford CoreNLP (Manning et al., 2014) が利用しやすいシステムとして言語処理のデファクトスタンダードとなりつつある。この状況を考慮し、我々は日本語処理の独自性をもとに、 Stanford CoreNLP の設計を踏襲しつつさらに拡張が容易な簡易フレームワークを構築している。このフレームワークの上に様々なツールのラッパーを用意することで、例えばユーザーは、同じレイヤーの出力を与える異なるシステムを透過的に用いることができる。例として文節単位の係り受けの上にシステムを構築したい場合、その情報を CaboCha から得るか KNP から得るかは一行の設定により変更が可能となり、どちらも同じ形式の XML で受け取ることができる。なお、本システムは本稿の公開までにオープンソースとして公開する予定である。

また Jigg は Scala（または JVM ）ベースの様々な日本語処理ソフトウェアの集合としての側面も持っている。例として、既に実装済みである日本語 CCG パーザを紹介し、現在の精度などを報告する。

2 基本設計と特徴

本ツールの目標は、設計をシンプルに抑えることで導入の敷居を低くし、多様なユーザーが容易に扱うことのできる NLP パイプラインを提供することである。UIMA (Ferrucci and Lally, 2004) などの既存のツールは多様な機能を備える代わりに導入の敷居が大きく、ユーザーはツールの学習に多くの時間を取られてしまう。Stanford CoreNLP (Manning et al., 2014) は、機能を単一マシンでの生テキストからの単純なパイプライン処理に制限することで、ユーザーにとって利用しやすいシステムを実現している。Jigg は Stanford CoreNLP とほぼ同じインターフェースを備えるため、日本語処理のための CoreNLP と見なすことができる。本節ではイメージを掴むためにまず簡単な使用例から始め、その後設計の方針を Stanford CoreNLP と対比しながら説明する。

2.1 文分割から CCG パーザまで

本節では我々の構築した日本語 CCG パーザ がパイプラインを通じてどのように利用可能になるかを述べる。本パーザの詳細は 4 節にまとめた。以下は生の文から CCG の解析結果を得る例である。

```
$ echo "東京は晴れです。 京都はどうですか。" | \
  java -cp "./jar/*" jigg.pipeline.Pipeline \
  -annotators ssplit,mecab,ccg > annotated.xml
```

この結果は以下のように XML で得られる。

```
$ cat annotated.xml
<?xml version='1.0' encoding='UTF-8'?>
<root>
  <document>
    <sentences>
      <sentence id="s0">
        東京は晴れです。
        <tokens>
          <token ... pos2="地域" pos1="固有名詞" pos="名詞" surf="東京" id="s0_t0"/>
          <token ... pos2="*" pos1="係助詞" pos="助詞" surf="は" id="s0_t1"/>
          <token ... pos2="*" pos1="一般" pos="名詞" surf="晴れ" id="s0_t2"/>
          <token ... pos2="*" pos1="*" pos="助動詞" surf="です" id="s0_t3"/>
          <token ... pos2="*" pos1="句点" pos="記号" surf="。" id="s0_t4"/>
        </tokens>
        <ccg score="468.7" id="s0_ccg0" root="s0_sp0">
          <span child="s0_sp1 s0_sp8" category="S[mod=nm,form=base]" ... id="s0_sp0"/>
          <span child="s0_sp2 s0_sp5" category="S[mod=nm,form=base]" ... id="s0_sp1"/>
          ...
          <span terminal="s0_4" category="S\S" ... id="s0_sp8"/>
        </ccg>
      </sentence>
      <sentence id="s1">
        京都はどうですか。
        <tokens>
          <token ... pos2="地域" pos1="固有名詞" pos="名詞" surf="京都" id="s1_t0"/>
          <token ... pos2="*" pos1="係助詞" pos="助詞" surf="は" id="s1_1"/>
          ...
        </tokens>
        <ccg score="520.4854664802551" id="s1_ccg0" root="s1_sp0">
          <span child="s1_sp1 s1_sp10" category="NP[mod=nm,case=nc]" ... id="s1_sp0"/>
          ...
        </ccg>
      </sentence>
  </document>
</root>
```

結果の XML には、各文毎に mecab の出力結果をまとめたもの (tokens) や CCG の導出 (ccg) が付与される。

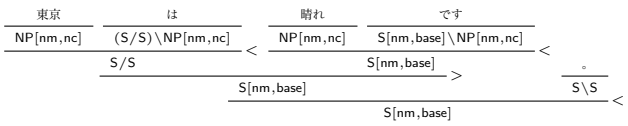

図 1: CCG の導出木。

図 1 に出力の一文目に対応する CCG の導出を示す。

引数の中で -annotators は必須であり、これにより入力に対して何の処理を行うかを決定する。この場合、パイプラインはまず与えられた入力を文に分割し (ssplit)、各文に形態素解析を行い (mecab)、その結果を CCG パーザに渡す (ccg)、という流れで解析を行う。多くのアノテータは、その動作をカスタマイズするために独自の引数を受け付ける。たとえば ssplit はデフォルトでは “。” もしくは改行で文を区切るが、

```
-ssplit.method newLine
```

と指定した場合、改行のみで文分割で行われ、上の入力に対しては分割が行われない。このような設定はコマンドラインを通じて指定ができるほか、次のようにプロパティファイルに記述し、これを指定しても良い。

```
$ cat sample.properties
ssplit.method: newLine
mecab.command: /home/jigg/bin/mecab
$ java -cp "./jar/*" jigg.pipeline.Pipeline \
  -annotators ssplit,mecab -props sample.properties
>
```

ここでは更に mecab.command によって MeCab の実行パスを指定している。またこのように入力ファイルを省略した場合は対話モードで起動する。

2.2 処理の流れ

このとき内部では次のような処理が行われている。

1. -annotators 引数で与えられたアノテータ間の依存関係を計算し、これらが順に実行可能なものであるかをチェックする。例えば mecab は事前に文分割が行われていることを必要とし、 ccg は形態素解析が行われていることを必要とする。
2. 各アノテータは Scala の XML オブジェクトを受け取り、それに情報を追加していく。例えば mecab は以下のような ssplit が返す XML に、tokens のアノテーションを追加する。

```
<root><document><sentences>
  <sentence id="s0">東京は晴れです。</sentence>
  <sentence id="s1">京都はどうですか。</sentence>
</sentences></document></root>
```

2.3 API としての使用例

以上はコマンドラインからの使用例であるが、本システムは Scala で実装されているため、Scala や Java から直接呼び出すことができる。図 2 に Scala からの

```
import jigg.pipeline.Pipeline
import scala.xml.Node
import scala.collection.mutable.HashMap
import java.util.Properties

object ScalaExample {
  def main(args: Array[String]): Unit = {

    val props = new Properties
    props.setProperty("annotators", "ssplit,mecab")

    // Pipeline クラスがアノテートを行う。この設定を Properties で行う。
    val pipeline = new Pipeline(props)

    // 文字列からアノテートを行う。結果は scala.XML.Node オブジェクトで得られる。
    val annotation: Node = pipeline.annotate("東京は晴れです。京都はどうですか。")

    val sentences: Seq[Node] = annotation \\ "sentence" // 再帰的に sentence を検索。
    val sentence: Node = sentences(0) // 東京は... の XML オブジェクト

    sentence \ "tokens" \ "token" foreach { t => // 各 token element について
      val surface: String = (t \ "@surf").toString
      val pos: String = (t \ "@pos").toString
      ... // 表層と品詞を取り出して何か処理を行う。
    }
  }
}
```

図 2: Scala API としての利用例。

```
Node annotation = pipeline.annotate("Java からも使えます。");

// Java では \\ などが使えないが、ヘルパー関数で検索が行える。
List<Node> sentences = XMLUtil.findAll(annotation, "sentence");

Node sentence = sentences.get(0);
Node tokens = XMLUtil.find(sentence, "tokens");
for (Node token : XMLUtil.findAll(tokens, "token")) {
  String surface = XMLUtil.find(token, "@surf").toString();
  ...
}
```

図 3: Java からの使用例。

利用例を示す。このように、アノテートされた結果は、Scala の XML オブジェクトである scala.xml.Node クラスのインスタンスとして渡され、Scala の組み込みメソッド \ と \\ などで直感的に内部データにアクセスすることができる。これらのメソッドは Java から用いることができないが、代わりに jigg.util.XMLUtil に定義された findAll などのヘルパー関数を用いることで検索を行うことができる。図 3 にその例を示す。

2.4 Stanford CoreNLP との違い

以上が基本的な使い方であるが、このように Jigg は Stanford CoreNLP とほぼ同一のインターフェースを採用している。一番の大きな違いは、各アノテータが情報を追加するオブジェクトである。Stanford CoreNLP は CoreMap と呼ばれる独自のオブジェクトを操作し、ここに各アノテータが情報を加えていくのに対し、 Jigg では各アノテータが Scala の XML オブジェクトを操作する。このような仕様にしたのは、主に拡張性の理由による。Stanford CoreNLP はユーザーが独自にアノテータを追加することができるが、その使途は限定されている。複雑な情報を追加するアノテータを実装しようとすると、そのような情報を保持するクラスも実装する必要がある。また一部のコード、例えば CoreMap を XML で出力するコードは独自のクラスに対応できるようになっていないため、文節や KNP の基本句など Stanford CoreNLP が想定してい

る英語処理の枠組みに収まらないオブジェクトは簡単に扱うことができない。そこで Jigg では、各アノテータは XML を直接参照し変更できるようにした。この XML は文字列ではなくオブジェクトなので、メモリや実行速度の点でそれほど大きな問題にはならない。これにより各アノテータは大きな自由度を持ち、新しいアノテータの実装がより容易になる。

その他の基本方針は Stanford CoreNLP とほぼ同一である。我々は機能の豊富さよりも扱いやすさを重視し、また速度もそれほど重視しない。XML 処理の多少のオーバーヘッドは、文単位の並列処理の機能をサポートすることで解消されると考えている。日本語処理にはまた、ソフトウェア毎の辞書の取り扱いの違いなど独自の問題があるが、これについては次節で触れる。

3 依存関係のチェック

パイプラインは、まず与えられたアノテータが順に正しく実行できるかをチェックする。例えば CCG パーザは形態素解析のアノテーションが済んでいることを前提とするし、KNP は前段階として JUMAN による解析を必要とする。基本的な方針として、まず各アノテータクラスは、それを実行する前に必要とするアノテーションの識別子 (requires) 及びそれにより追加されるアノテーションの識別子 (requirementsSatisfied) を持つ。もし requires が全てそれまでのアノテータにより与えられていれば、チェックは完了する。例えば文分割のアノテーションが済んだことを表す識別子である Requirement.Sentence は ssplit により与えられ、mecab などはこれを必要とする。

ここで日本語特有の問題として、辞書の違いによる要件の複雑化が挙げられる。例えば現在の CCG パーザは IPA 辞書で構築されたコーパスをもとに学習されているため、JUMAN の解析結果を使うことができない（juman,ccg と指定すると失敗することが望ましい)。一つの方針は、辞書毎の形態素解析や係り受け解析の結果を完全に区別し、依存関係に辞書への依存も持たせることである。しかしながら、場合によってはどの辞書を使ったかには関知せず、単に何らかの単語分割の結果に依存したアノテーションを行うことも多いだろう。そのため、現在の方針としては、各アノテーションの識別子に階層関係を持たせるように実装を進めている。例えば MeCab の IPA 辞書モデルで解析した場合の識別子は、TokenizeWithIPA で与えられるが、これはより一般の識別子である Tokenize と階層関係を持ち、別のアノテータが Tokenize を必要とする場面では、TokenizeWithIPA が代わりに満たされていても良い。しかし TokenizeWithJuman が必要とされるケースでは TokenizeWithIPA や Tokenize は代わりに用いることができない。

```
<sentence id="s0">
  太郎はケーキを買って食べていた
  <tokens>
    <token id="s0_t0" surf="太郎" pos="名詞" pos1="人名" ...
      features="&quot; 人名:日本:名:45:0.00106 疑似代表表記 代表表記:太郎/たろう..."/>
    <token id="s0_t1" ... features="NIL &lt; かな漢字&gt;&lt; ひらがな&gt;&lt;..."/>
    ...
  </tokens>
  <basic_phrases>
    <basic_phrase id="s0_bp0" tokens="s0_t0 s0_t1" features="&lt; 文頭&gt;&lt; 人...">
    ...
  </basic_phrases>
  <chunks>
    <chunk id="s0_chu0" tokens="s0_t0 s0_t1" features="&lt; 文頭&gt;&lt; 人名...">
    ...
  </chunks>
  <basic_phrase_dependencies root="s0_bp3">
    <basic_phrase_dependency id="s0_bpdep0" head="s0_bp3" dependent="s0_bp0" label="D"/>
    ...
  </basic_phrase_dependencies>
  <dependencies root="s0_chu3">
     <dependency id="s0_dep0" head="s0_chu3" dependent="s0_chu0" label="D"/>
     ...
  </dependencies>
  <case_relations>
    <case_relation id="s0_cr0" head="s0_bp2" depend="unk" label="ガ" flag="U"/>
    <case_relation id="s0_cr1" head="s0_bp2" depend="s0_tok2" label="ヲ" flag="C"/>
    ...
  </case_relations>
  <coreferences>...</coreferences>
  <predicate_argument_relations>
    <predicate_argument_relation
      id="s0_par0" predicate="s0_bp2" argument="s0_coref9" label="ヲ" flag="C"/>
    ...
  </predicate_argument_relations>
  ...
</sentence>
```

図 4: KNP の解析結果の例。dependencies など一部は CaboCha と構造を共有する。

4 既存のアノテータ

現在以下のようなアノテータの利用をサポートしている。ここでは -annotators に渡す形の省略形で示す。

- ssplit: 文分割を行う。必ず最初に呼ばれることを前提としている。内部では正規表現で文分割を行っており、2.1 節で示したような挙動の変更の他、正規表現を直接与えることもできる。
- mecab, kuromoji: 形態素解析を行う。IPA 辞書以外の扱いに関しては現在検討中である。mecab などの外部ツールは全てプロセス間通信を行うラッパーで実装している。
- cabocha: 形態素解析された各文に対し、文節情報と文節間の係り受け関係の情報をアノテートする。
- juman: JUMAN の出力を mecab などと同じ形式で出力する。曖昧な解析は token_alt 要素によって表現され、KNP に渡される。
- knp: 各文毎に juman が出力する XML による解析結果から JUMAN のもとの出力を復元し、文節や基本句の係り関係、固有表現、述語項構造などの KNP の出力 (笹野ら, 2013) を XML にまとめる。図 4 に、knp の出力例を示す。

```
package my_project

// 新しい SentencesAnnotator。各文の全ての名詞 n グラムに対し、それが companyNames に
// 含まれていたら、新しく company 要素のアノテーションを追加する。
class CompanyNameAnnotator(companyNames: Set[String]) extends SentencesAnnotator {

  // (String, Properties) を受け取るコンストラクタを用意しておくと、コマンドラインからこの
  // クラスのパスを指定することで呼び出すことが可能になる。
  def this(name: String, props: Properties) =
    // -company.list xxx と与えられたパスから集合を得てコンストラクタに渡す。
    this(CompanyNameAnnotator.readList(props.getProperty("company.list")))

  // SentenceAnnotator はこのメソッドを実装する必要がある。annotate メソッドは内部で
  // このメソッドを呼び出す。現在の sentence ノードが渡されるので、新しい情報を追加して返す。
  override def newSentenceAnnotation(sentence: Node) = {
    val tokens = sentence \ "tokens" \ "token"
    val hitEntities: Seq[Node] = allNounChunks(tokens) flatMap { case (begin, end) =>
      val name = (begin to end) map(i=>(tokens(i) \ "@surf").toString) mkString("")

      // allNounChunks をもとに結合した名詞句が companyNames の中にあれば
      companyNames.find(_ == name) map { _ =>
        val beginID = (tokens(begin) \ "@id").toString
        val endID = (tokens(end) \ "@id").toString
        // その XML ノードを作る。
        <company id={nextID} begin={beginID} end={endID} name={name} />
      }
    }
    // sentence 要素に新しく companies を追加する。
    XMLUtil.addChild(sentence, <companies>{hitEntities}</companies>)
  }
  // 呼び出される毎に新しい ID を付与。
  private[this] var entityID = -1
  def nextID: String = { entityID += 1; "c" + entityID }

  // 全ての名詞の連接を見つけて、始まりと終わりのインデックスの組を返す。
  private[this] def allNounChunks(tokens: Seq[Node]): Seq[(Int, Int)] = ...

  // 事前に形態素解析が行われていることを必要とする。
  override def requires = Set(Requirement.Tokenize)
}
```

図 5: アノテータの実装例。

- ccg: 我々は Uematsu et al. (2013) の構築した日本語 CCG Bank をもとに高精度な日本語 CCG パーザを実装した。本パーザからは、本来 CCG の単語間長距離依存構造を得ることができるが、現時点では図 1 に示したような木構造の導出のみに限られる。内部はビームサーチを組み合わせた Shift-reduce パーザとして実装されている (Zhang and Clark, 2011)。出力を文節単位の係り受けに変換した際の京大コーパスに対する精度は約 88% と、CaboCha などと比べると高くないが、現時点で数少ない日本語の句構造パーザとして活用できる可能性がある。

5 新しいアノテータの追加

新しいアノテータを実装する際の手順について簡単にまとめる。まず全てのアノテータは、 jigg.pipeline.Annotator クラスのサブクラスである必要がある。これは全てのアノテータが次の annotate メソッドを実装する必要があることを意味する。

```
def annotate(annotation: Node): Node
```

その他、3 節で述べた次のメソッドは、パイプラインの中で依存関係を解決させる際に必要である。

```
def requires: Set[Requirement]

def requirementsSatisfied: Seq[Requirement]
```

図 5 に示した例は、Annotator ではなく SentencesAnnotator を継承している。これは Annotator のサブクラスであり、cabocha や knp など多くの既存アノテータがこれに属する。図 5 のように、この場合実装すべきメソッドは annotate の代わりに newSentenceAnnotation となる。この新しいアノテータは辞書ベースの簡単なマッチングにより、文中で会社名にマッチする n グラムを探し、あればその情報をアノテートする。このアノテータを使う方法はいくつかあるが、一つは図 5 のように (String, Properties) を受け取るコンストラクタを定義する方法である。こうすると、次のようにコマンドラインからの実行時にクラスパスを直接指定することでクラスのインスタンス化がリフレクションを用いて行えるようになる。

```
$ java -cp "./jar/*" jigg.pipeline.Pipeline \
  -annotators ssplit,mecab,my_project.CompanyNameAnnotator \
  -company.list companies.txt
```

また既存アノテータのように省略名を登録することもできる。このようなコンストラクタは単にアノテータを使う場合には必須ではない。例えば次のように、既存クラスによるアノテーションが済んだ後の XML に対して annotate を直接実行しても良い。

```
val myAnnotator = new CompanyNameAnnotator(companyNames)
val partialAnnotation = pipeline.annotate(input)
val annotation = myAnnotator.annotate(partialAnnotation)
```

6 おわりに

日本語処理を行いやすくするためのフレームワーク Jigg の紹介を行った。本システムは実装済みの解析器であればコマンドラインから簡単に繋ぐことができるほか、Java や Scala を通じて拡張を行うこともできる。JVM 上のシステムは maven を通じたダウンロードや一つの jar ファイルで完結するという点で配布しやすいという利点があるが、他の言語から利用できることも重要であり、今後より多くの人に使いやすい環境を目指して開発を進めていきたい。

謝辞

本研究は JST CREST から助成を受けた。

参考文献

D. Ferrucci and A. Lally. Uima: An architectural approach to unstructured information processing in the corporate research environment. *Nat. Lang. Eng.*, 10(3-4):327–348, 2004.

C. D. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. J. Bethard, and D. McClosky. The Stanford CoreNLP natural language processing toolkit. In *Proc. of ACL: System Demonstrations*, pages 55–60, 2014.

S. Uematsu, T. Matsuzaki, H. Hanaoka, Y. Miyao, and H. Mima. Integrating multiple dependency corpora for inducing wide-coverage japanese ccg resources. In *Proc. of ACL*, pages 1042–1051, 2013.

Y. Zhang and S. Clark. Shift-reduce ccg parsing. In *Proc. of ACL-HLT*, pages 683–692, 2011.

笹野 遼平, 河原 大輔, 黒橋 禎夫, 奥村 学. 構文・述語項構造解析システム knp の解析の流れと特徴. 言語処理学会 第 19 回年次大会, pages 110–113, 2013.