Vol.2014-NL-219 No.12
2014/12/17

ソースコード構文木からの統計的自動コメント生成

小田 悠介[1,a)]　札場 寛之[2,b)]　ニュービッグ グラム[1,c)]　サクティ サクリアニ[1,d)]　戸田 智基[1,e)]
中村 哲[1,f)]

概要：プログラミング初学者にとって，与えられたソースコードがどのような処理内容なのかを把握するのは容易ではない．そこでソースコード読解支援のために，与えられたソースコードから処理内容を示すコメントを自動的に生成し，ソースコードと共に提示することで読解を促すシステムが考えられる．本研究ではコメント生成のために Tree-to-String 統計翻訳の枠組みを使用し，プログラミング言語の構文木とコメントに対して翻訳器を学習することで，ソースコードから統計的にコメントを生成するシステムを提案する．

1. はじめに

プログラミングを行う者にとって, ソースコードの読解は必須の技能の一つである．使用するプログラミング言語の基本的な文法を理解し, ソースコードに書かれた動作を素早く正確に把握できるようになることは, グループでの共同開発作業やプログラミング学習の面で重要である．この中で, プログラミング熟練者が大規模な共同開発プロジェクトの精度を把握するためのツール [1] や, ソースコードが期待通りに動作しない場合の解決手段を提示するシステム [2] などが提案されている．

しかし, ソースコードの読解者がプログラミング初学者である場合や, 読解者に馴染みのない言語で書かれたソースコードを読解しようとする場合, 読解者がその言語特有の文法や慣習についての知識を持っていないため, ソースコード 1 行ずつを読解すること自体に時間がかかると考えられる．

そこで, 基礎的なプログラミング読解の支援として, 与えられたソースコードを解析して処理内容に相当する自然言語の文を生成し, ソースコードのコメントとして適宜読解者に提示する方法が考えられる．具体的には図 1 に示すように, 任意のソースコード 1 行を読み取り, この行の表す処理内容を自然言語で出力するシステムを考える．

```
 input 1 (Python) :  if x % 5 == 0:
output 1 (Comment):  # x が 5 で割り切れるなら
 input 2 (Python) :      y = x / 5
output 2 (Comment):      # y に x と 5 の商を代入
```

図 1　Python ソースコードからの行単位コメント生成

与えられたソースコードにコメントを付与するシステムはソフトウェア工学の分野でいくつか提案されており, ルールに基づくコメント生成手法 [3], [4], [5] と情報検索に基づくコメント提示手法 [6] に大別される．

ルールに基づくコメント生成手法はソースコードとコメントの対応関係を直接考慮するため, ソースコードと密接に関係したコメントの生成が可能である．しかしシステムの構築・改良には人手によるルールの作成を必要とする上, 既存のシステムでは識別子 (変数名, 関数名などのプログラム要素に対する名前) が人間に分かりやすい形で記述されていることを前提としていたり, 特定のプログラム構造のみにコメントの生成対象を限定していたりするなど, 広い文脈でコメント生成が可能なシステムは構築されていない．逆に情報検索に基づく手法では大量のデータを使用して幅広いソースコードに対応することが可能であるが, 検索データベースに入力とするソースコードに対応するコメントが存在しない場合, 新たに適切なコメントを生成することは不可能である．

本研究では, ルールに基づく手法が行うようなコメント生成を, 人手によるルールの構築を介さず, データを用いた機械学習によって実現することを目指す．具体的には, ソースコードとその各行にコメントの付与されたデータを対訳

[1] 奈良先端科学技術大学院大学
Nara Institute of Science and Technology
[2] 信州大学
Shinshu University
[a)] oda.yusuke.on9@is.naist.jp
[b)] delihiros@gmail.com
[c)] neubig@is.naist.jp
[d)] ssakti@is.naist.jp
[e)] tomoki@is.naist.jp
[f)] s-nakamura@is.naist.jp

データと見なし, Tree-to-String 翻訳と呼ばれる統計的機械翻訳の枠組みによって学習することで, ソースコード構文木からコメント文字列への翻訳器を構築する. ここで, ソースコード構文木は自然言語の構文木とは異なる構造を持つため, 自然言語間の翻訳のために提案された Tree-to-String 翻訳のアルゴリズムを直接適用するには適さない. この問題を解決するために, ソースコード構文木を前処理によって自然言語の構文木に近づける変形を行う手法を提案する.

実験的評価では Python ver.3 で書かれたソースコードの各行に対して日本語のコメントを自動的に付与するコメント生成器を構築した. これによって生成されるコメントが人手によるコメントに近い文であるかどうか, またソースコードの内容をコメントが忠実に反映しているかどうかを機械翻訳の評価手法に基づいて評価した.

2. 統計的機械翻訳

2.1 統計的機械翻訳の枠組み

統計的機械翻訳 [7] では, 原文 $\boldsymbol{f}$ が与えられた下で最適な翻訳結果 $\hat{\boldsymbol{e}}$ を式 (1) の条件付き確率による推定問題として考える.

$$\hat{\boldsymbol{e}} := \arg\max_{\boldsymbol{e}} \Pr(\boldsymbol{e}|\boldsymbol{f}) \tag{1}$$

ここで $\boldsymbol{f} = f_1^{|\boldsymbol{f}|} = [f_1, f_2, \cdots, f_{|\boldsymbol{f}|}]$ は原文を表す単語列, $\boldsymbol{e} = e_1^{|\boldsymbol{e}|} = [e_1, e_2, \cdots, e_{|\boldsymbol{e}|}]$ は翻訳候補となる目的言語の単語列である. $|\boldsymbol{f}|$ 及び $|\boldsymbol{e}|$ はそれぞれの単語数を表す.

統計的機械翻訳の手法には, $\boldsymbol{f}$ のフレーズ (部分的な単語列) ごとの翻訳と並べ替えを行うフレーズベース翻訳 [8], フレーズの階層構造を考慮する階層的フレーズベース翻訳 [9], 原文の構文構造を考慮する Tree-to-String 翻訳 [10] などがある. 本研究では Tree-to-String 翻訳を用いるため, 次節で詳しく解説する.

2.2 Tree-to-String 翻訳

原文 $\boldsymbol{f}$ に関して単語列よりも詳しい情報を持つ構造が使用できる場合, その情報を考慮して翻訳を行うのが有効であると考えられる. このような構造の一つとして原文に関する構文木 $T_{\boldsymbol{f}}$ が挙げられる. 構文木は単語やフレーズの種別や相互の関係を特定の規則に基づく木構造で表現したものであり, 構文解析により単語列からその構造が推定される. 構文木に含まれる情報を参考にしながら目的言語の文を生成することで, 単語列のみを考慮する翻訳手法よりも翻訳精度を向上できることが知られている [11]. このような翻訳手法を Tree-to-String 翻訳と呼ぶ.

形式的には, 式 (1) に構文木に関する周辺化を導入することで, Tree-to-String 翻訳を定式化できる.

$$\begin{aligned}
\hat{\boldsymbol{e}} &:= \arg\max_{\boldsymbol{e}} \Pr(\boldsymbol{e}|\boldsymbol{f}) \\
&= \arg\max_{\boldsymbol{e}} \sum_{T_{\boldsymbol{f}}} \Pr(\boldsymbol{e}|\boldsymbol{f}, T_{\boldsymbol{f}}) \Pr(T_{\boldsymbol{f}}|\boldsymbol{f}) \\
&\simeq \arg\max_{\boldsymbol{e}} \sum_{T_{\boldsymbol{f}}} \Pr(\boldsymbol{e}|T_{\boldsymbol{f}}) \Pr(T_{\boldsymbol{f}}|\boldsymbol{f}) \qquad (2) \\
&\simeq \arg\max_{\boldsymbol{e}} \Pr(\boldsymbol{e}|T_{\boldsymbol{f}}^*) \qquad (3)
\end{aligned}$$

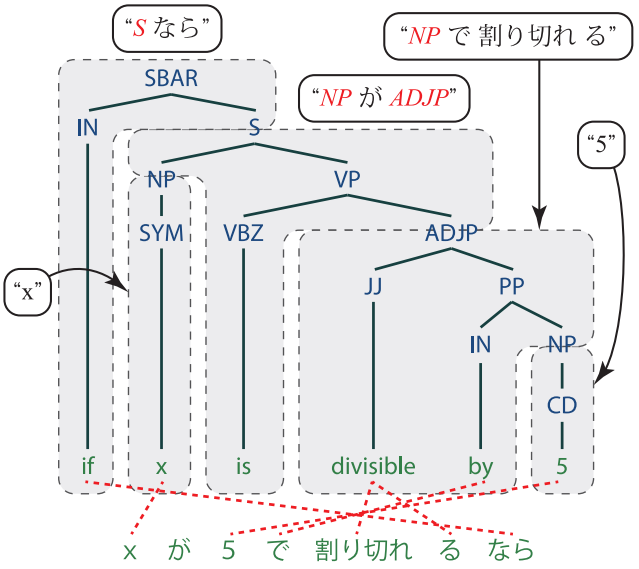


図 2 Tree-to-String 英日翻訳のルールとアライメント

Tree-to-String 翻訳では, 図 2 に示す英日翻訳の例のように, 入力された原文の構文木を部分木に分割し, 各部分木を目的言語のフレーズに置き換えることで翻訳を行う. ここで図 2 の英語側の構文木は句構造と呼ばれ, 原文の構造を文脈自由文法により表現したものである. 部分木と目的言語のフレーズの組をルールと呼び, ルールの集合を導出 $\boldsymbol{d}$ と呼ぶ. $\boldsymbol{d}$ が定まれば翻訳結果 $\boldsymbol{e}$ は一意に決まるため, Tree-to-String 翻訳は入力構文木に対する最適な導出を選択する問題とも考えることができる.

$\Pr(\boldsymbol{e}|T_{\boldsymbol{f}})$ は原文の構文木から目的言語による翻訳文が生成される確率であり, 式 (4) に示す $\boldsymbol{d}$ を用いた対数線形モデルで表現される.

$$\Pr(\boldsymbol{e}|T_{\boldsymbol{f}}) := \frac{\sum_{\boldsymbol{d}\in\mathcal{D}_{\boldsymbol{e}}} \exp\left(\boldsymbol{w}\cdot\boldsymbol{\phi}(T_{\boldsymbol{f}}, \boldsymbol{e}, \boldsymbol{d})\right)}{\sum_{\boldsymbol{e}}\sum_{\boldsymbol{d}\in\mathcal{D}_{\boldsymbol{e}}} \exp\left(\boldsymbol{w}\cdot\boldsymbol{\phi}(T_{\boldsymbol{f}}, \boldsymbol{e}, \boldsymbol{d})\right)} \tag{4}$$

ここで $\mathcal{D}_{\boldsymbol{e}}$ は $\boldsymbol{e}$ を翻訳結果として生成するような導出の集合, $\boldsymbol{\phi}$ は $\boldsymbol{d}$ を特徴付ける素性関数, $\boldsymbol{w}$ は素性に関する重みベクトルである. $\boldsymbol{\phi}$ の要素としては $\boldsymbol{e}$ の自然性を表す言語モデルや部分木の複雑さ, 部分木が目的言語のフレーズを生成する確率などが用いられる.

$\Pr(T_{\boldsymbol{f}}|\boldsymbol{f})$ は原文 $\boldsymbol{f}$ から構文解析により構文木 $T_{\boldsymbol{f}}$ を生成する確率である. 自然言語の場合, 構文木は一意に定まるものではなく, 文自体の曖昧性により複数の構文木が得られる可能性がある. 式 (3) に示すように, Tree-to-String 翻訳では可能な構文木の中で尤度最大のもの $T_{\boldsymbol{f}}^* = \arg\max_{T_{\boldsymbol{f}}} \Pr(T_{\boldsymbol{f}}|\boldsymbol{f})$ ただ一つを用いて翻訳を行い, 構文木の生成確率については考慮しない. このため, 構文解

析の曖昧性が大きい文に対しては誤った翻訳結果を生成してしまう可能性がある．この問題を解決するために，式 (2) のように構文木の多様性を考慮する翻訳手法も提案されている [12]．

2.3 Tree-to-String 翻訳器の学習

Tree-to-String 翻訳器は (1) 終端記号のアライメント学習，(2) ルールの抽出，(3) 素性の重み学習を行うことにより構築される．

2.3.1 終端記号のアライメント学習

句構造などの自然言語処理で用いられる構文木は，終端記号に解析対象となった文の単語が並んでいる．この記号列と目的言語の単語との間には一定の対応関係があると考えられる．この対応関係をアライメントと呼び，図 2 では赤破線で示されている．アライメントは人手による付与も可能だが，対訳データから教師なし学習により統計的に求める手法 [13], [14], [15] が一般的に用いられる．

2.3.2 ルール抽出

得られたアライメントを参考にして，対訳データからルールの候補を抽出する．これには GHKM アルゴリズム [16] などのヒューリスティクスが用いられる．また，抽出されたルールのから素性 ϕ の計算に必要な統計量を算出し，ルールに対応付けておく．

2.3.3 素性の重み学習

抽出されたルールの集合と事前定義した素性関数を使用して，重み学習用の対訳データに対する翻訳精度が高くなるような重みベクトル $\boldsymbol{w}$ の学習を行う [17]．

3. ソースコード構文木からの自動コメント生成

本節では，Tree-to-String 翻訳を用いてソースコード構文木からコメントを自動的に生成する手法を提案する．

自動コメント生成における入力データは自然言語ではなく，特定のプログラミング言語で記述されたソースコードである．ソースコードが最終的にプログラムによって解析されることを目的とする関係上，実用的なプログラミング言語はコンパイラやインタプリタによる構文解析が容易になるよう言語仕様が設計されている．このため，ソースコードの構文解析結果は基本的に一意に定まり，自然言語を入力とする従来の機械翻訳とは異なり構文解析誤りによる悪影響がないと考えてよい．

このため，自動コメント生成に用いる翻訳手法としては Tree-to-String 翻訳が適していると考えられる．以降の節では Tree-to-String 翻訳を用いた自動コメント生成法の詳細を述べる．

3.1 コメント生成の対象とソースコードの構文解析

先行研究ではソースコードの文単位でコメントの付与を行う手法と，一定のまとまりに対してコメントの付与を行う手法が存在する．本研究が対象とするのは文単位の読解支援であるため，コメントの生成対象はソースコードの各行とする．

Python では基本的に改行が文区切りの役割を果たすため，ほとんどのソースコードは行単位で切り出して独立に構文解析することが可能である．ただし，`if` 文や関数定義などの制御構造に関する構文を中心に，複数行による記述を意図している文法が一部存在する．切り出した行がこのような文に該当する場合は単独で解析できないため，文に対して適切なトークンを補うことで構文解析時の整合性を取る必要がある．具体的には，バックスラッシュ文字による強制改行を全て除去した後，表 1 に示すようなルールを各行に適用してから構文解析を行い，解析結果からルールによる変更箇所に対応する部分木を除去することで構文木を得ることができる．

3.2 プログラミング言語の構文木の特徴

3.1 節で述べた手法で構文木を生成すれば，Tree-to-String 翻訳器の学習は可能である．しかし，自然言語の構文木とプログラミング言語の構文木の間には大きな差があり，これに起因する問題に対処する必要がある．

例えば図 3 に示すのは，英語の文「if x is divisible by 5」に対する句構造木 (a)，及び Python の「`if x % 5 == 0:`」に相当する構文木を句構造と同様の形式に整形したもの (b) である．英語の構文木は句構造解析器 Ckylark[*1] による推定結果であり，Python の構文木は標準ライブラリである `ast` を使用して生成した．これらの構文木にはどちらも日本語訳として「x が 5 で割り切れるなら」という文が考えられる．

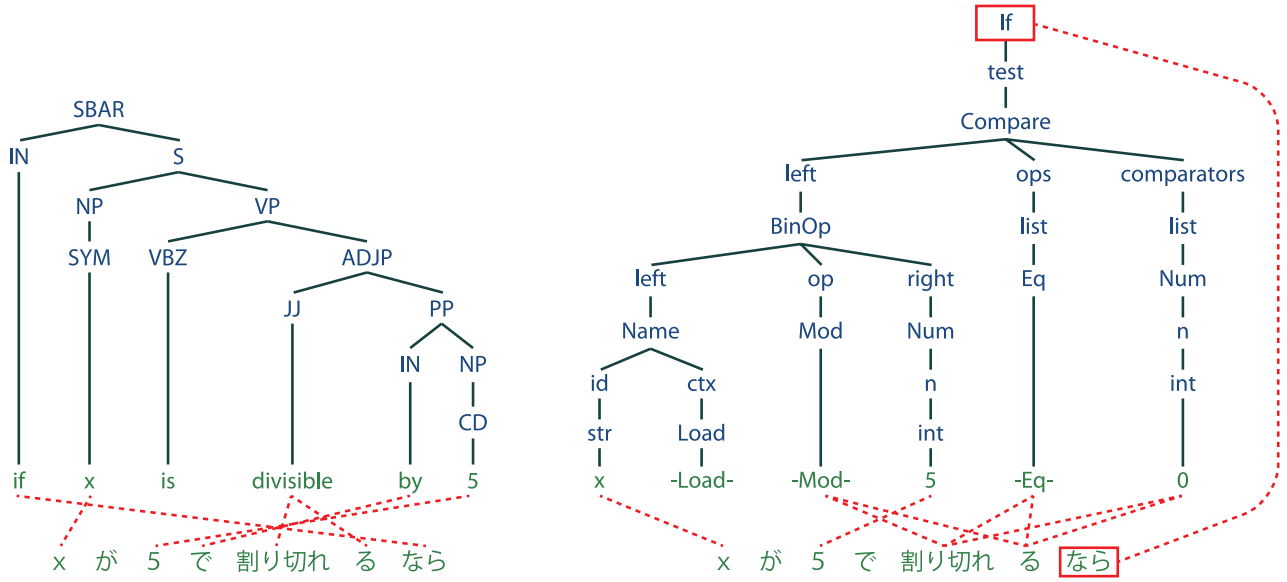
まず，Python の構文木には終端記号と原文のトークンの一部に対応関係がないことが見受けられる．例えば図 3(b) の `Load` 句は変数の読み込みを指示する構造であるが，原文には直接存在せず，構文解析の過程で解析器に付加されたものである．また原文の「`if`」は構文木上では非終端記号として存在しており，終端記号としては現れない．

自然言語間の Tree-to-String 翻訳では 2.3 節で述べたように，まず原文の構文木の終端記号と正解訳の間のアライメントを自動的に学習し，この情報を基にして翻訳ルールの構築を行う．図 3 の破線は理想的なアライメントを示しているが，終端記号のみを対象としてアライメントを取る枠組みでは，図 3(b) のような非終端記号が目的言語側と対応付けられる状況には対応できない．また，`Load` 句のように実際にはアライメントの取られるべきでない記号が多数存在すると，これがノイズとなって誤ったアライメントを学習してしまう可能性が高くなる．

[*1] http://odaemon.com/?page=tools_ckylark

表 1　Python ソースコード各行に対する前処理の例

パターン	変換処理	原文の例	変換後の例
末尾のトークンが":"	末尾に"pass"を追加	`if a == b:` `def foo(a, b, c):`	`if a == b:pass` `def foo(a, b, c):pass`
先頭のトークンが"elif" または"else"	先頭に"if True:pass\n"を追加	`elif foo():` `else:`	`if True:pass\nelif foo():pass` `if True:pass\nelse:pass`
先頭のトークンが"except"	先頭に"try:pass\n"を追加	`except:`	`try:pass\nexcept:pass`



(a) 自然言語同士（英語→日本語）　(b) プログラミング言語と自然言語（Python→日本語）

図 3　自然言語とプログラミング言語における構文木とそのアライメントの違い

更に, 様々なソースコードの形式に対応するために構文木自体が冗長な構成になっており, これも学習時のノイズとなることが考えられる.

自然言語間の機械翻訳においても, 翻訳対象の言語対によっては構文解析器によって推定された原文の構文木が必ずしも翻訳に適した構造であるとは限らない. この問題に関して, 構文木を生成したい言語に合わせて翻訳しやすい構造に変換することで, 翻訳精度を向上させられることが報告されている [18], [19], [20].

従って, Tree-to-String 翻訳で生成されるコメントの精度をより高くするためには, ソースコードから生成された構文木をより翻訳アルゴリズムに適した形式に変換することにより, これらの不都合を取り除く必要があると考えられる. 次節では本研究で実際に用いた変換ルールを述べる.

3.3　コメント生成のためのソースコード構文木変換

コメント生成のための構文木変換として, Python の構文木に対する処理を説明する. 具体的には (1) **主辞の追加**, (2) **ルールによる構文木の簡略化**の 2 種類の最適化手法と, コメント文を参考にした (3) **識別子と定数の抽象化**を提案する.

3.3.1　主辞の追加

図 3(b) の `if` 句のように, 具体的な意味を持つ記号が構文木の非終端記号となっている場合に対応するには, 構文木の終端記号以外についてもコメントに現れる単語との対応関係を学習する必要がある. これについて, 自然言語の意味解析では非終端記号を特定の順で並べた列に対して学習を行う手法が提案されている [21]. ここではより一般化した手法として, 非終端記号の主辞に相当する記号を構文木に追加する操作を提案する.

主辞を追加する位置についてはフレーズの先頭または末尾, 適切な兄弟要素の間など議論の余地があり, 最適な追加位置は生成されるコメントの言語に依存する. 文献 [21] では英語との対応を取るために主辞をフレーズの先頭へ追加している. 本研究が対象とする日本語は主辞後置型の言語であり, 動作などを表す語は一般的にフレーズの末尾に位置する. このため, 本手法による主辞の追加位置もフレーズの末尾が適切であると考えられる.

図 3(b) の構文木に対して後置型の主辞を追加した木を図 4 に示す.

3.3.2　ルールによる構文木の簡略化

主辞の追加によって, 原理的には構文木の任意の部分に対するアライメントを学習することができるようになる. しかし図 4 は元の構文木に対して木の冗長さが増加しており, コメント生成に必要のない情報が多く残っている. また, 元の構文木本来の構造に変化を与えたわけではないた

表 2　構文木の簡略化ルールの例

パターン	変換後
(Name ... (id (str x)) ...)	(Name x)
(Num ... (n (int x)) ...)	(int x)
(BinOp ... (left x) (op y) (right z) ...)	(BinOp (left x) (right z) (op y))
(Compare ... (left x) (ops y) (comparators z) ...)	(Compare (left x) (comparators z) (ops y))

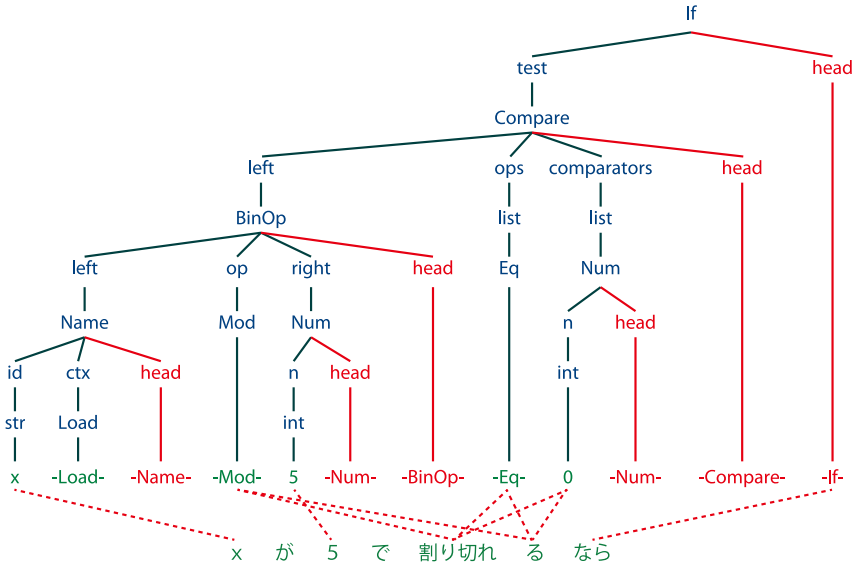


図 4　構文木への後置型主辞の追加

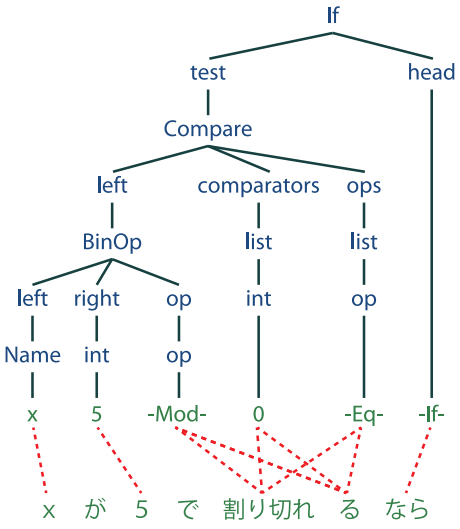


図 5　ルールによる構文木の簡略化

め, 演算子や識別子, 定数など元々存在する記号の順序に起因する誤りへの対処はされていない.

これらに対応するため, 主辞を追加した後の構文木に対してルールによる木構造の簡略化を行う. 本研究では, 木構造の簡略化に関するルール 20 個を人手により構築した. ルールの一部を表 2 に示す. また, 図 4 に簡略化ルールを適用した結果の構文木を図 5 に示す. 図 3(b) と図 5 の構文木に対するアライメントを比較すると, 後者は終端ノードに適切な記号が並んでおり, その順序も日本語に近いものとなっていることが分かる.

3.3.3　識別子と定数の抽象化

多くのプログラミング言語では言語仕様に抵触しない範囲で識別子を任意に命名することができ, 識別子に含まれる文字列に構文上の意味は含まれないと考えられる. また, 多くの文で整数や文字列といった定数項が使用されるが, 文脈によっては定数を任意の値に変更しても同様のコメントで問題ない可能性がある. 例えば「`if x % 5 == 0:`」に対して「x が 5 で割り切れるなら」というコメントが付いていれば, 変数名 x と定数 5 はソースコードとコメントの両方に含まれるため, 任意の別の値に置き換えても正しいコメントである可能性が高い. 一方定数 0 はコメントに明示的な対応はなく, 「割り切れる」というフレーズに暗黙的に含まれる値であることが考えられる.

3 で述べたように, ソースコードから生成された構文木には構文的な曖昧性が存在しないと仮定できるが, 上記のような置き換え可能な識別子や定数に関しては曖昧性が残っていることになる. この曖昧性を取り除くために, 識別子と定数の抽象化を構文木に対して適用する.

具体的な抽象化の手順を図 6 に示す. まず, 元の構文木からすべての識別子と定数を抽出し, それぞれに別名を付与する. 図 6 には識別子 x と定数 5, 0 を含んでいるため, それぞれに NAME1, INT1, INT2 という別名を付与する. 次に正解訳の単語列を探索し, 識別子については識別子名と文字列の一致するものを, 定数については値の一致するものを抽出する. こうして抽出された別名は抽象化可能な識別子や定数を表していると考えられるため, 構文木の終端記号とコメント中の識別子と定数を置き換え, 新たな構文木とコメントを生成する. このようにして抽象化を行うことで, 識別子や定数に関する曖昧性を大きく抑えることができると考えられる.

ただし, 構文木中に存在する識別子や定数が抽象化可能かどうかは正解のコメントが得られる学習時にしか調べられないため, 抽象化されたデータを用いた翻訳は通常の Tree-to-String 翻訳の枠組みで完全に対応することができない. これについては以降の章でより詳しく考察する.

3.4　統計的コメント生成器の学習

ソースコードとコメントの対訳データを用意することができれば, 以上の手法により変換されたソースコード構文木とコメントから 2.3 節で述べた手法を用いて Tree-to-String

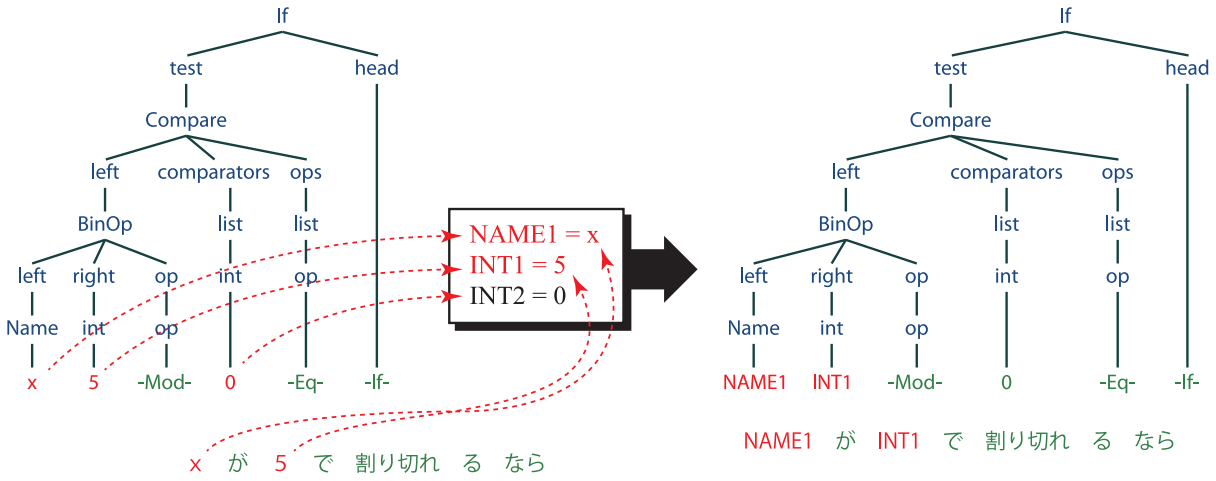


図 6　識別子と定数の抽象化

翻訳器を学習し, ソースコード構文木からの統計的コメント生成器を構築することができる. 次節では本研究の評価実験に用いた設定について詳しく述べる.

4. 評価実験: Python からの日本語コメント生成

4.1 学習データ

Tree-to-String 翻訳器を学習するためには, 原文と正解訳の組の集合である対訳コーパスが必要である. 今回は小田らが収集した Python ソースコード [22] を使用した. これは算術問題 [*2] に関する回答の Python ソースコードを 3 人のプログラマにより生成したものである. この内, 独立したコメント行及び巨大なデータを直接定義している行を取り除いた 722 行を抽出し, Python によるプログラミングが可能, かつ機械翻訳の知識を持たない 1 人のアノテータが各行にコメントを付与した.

アノテータが全データへのコメント付与に要した時間は約 4 時間であった. 次節で説明する Tree-to-String 翻訳器構築のための計算時間は 1 分程度であり, コメント付与にかかる時間に比べて十分に小さい. このため, 統計的コメント生成器を学習するのにかかる時間の大部分はソースコードとコメントの対訳コーパスの作成時間となる.

4.2 翻訳器の構築手順

図 7 に示すのは, 本研究における自動コメント生成システム構築の全体像である. 以下に示す手順で学習データの整形・学習を行い, 自動コメント生成のための Tree-to-String 翻訳器の構築を行った.

(1) 学習データの分割　与えられたデータをソースコード部分とコメント部分のテキストデータに分割する.

(2) 単語分割　日本語コメント文の単語分割 (分かち書き) を行い, 形態素列を得る. 本研究では単語分割ツールとして MeCab[23] を使用した. MeCab による分かち書き後, 生成された形態素列に ASCII 文字のみからなる形態素の連続があった場合, これを再結合して一つの形態素とした. これは「`foo _ bar`」「`1 e - 3`」といった形態素列はソースコード内に出現する識別子や定数が誤って分割されたものと考えられるためである.

(3) 構文解析　Python ソースコードに対して 3.1 に示した前処理を施し, 文単位の構文解析を行って構文木を得る. 構文解析には Python 標準ライブラリの `ast` を使用した.

(4) 構文木の最適化　(3) で得られた構文木について, 3.3 で提案した手法により最適化を行う.

(5) 構文木・形態素列の抽象化　(2) で得られた日本語の形態素列と (4) で得られた最適化済み構文木を用いて, 両方に共通して出現する識別子や定数の抽象化を行う.

(6) アライメント学習　(5) で得られた構文木から終端記号列を抽出し, これと形態素列との間のアライメントを学習する. アライメント学習には反転トランスダクション文法 (inversion transduction grammar: ITG) に基づく手法である pialign[15] を使用した. アライメント時の最大フレーズ長 (まとめてアライメントが生成される連続した単語列の最大長) は 10 とした. また, 単語数の多い文は学習ノイズとなりやすいため, 構文木の終端記号数かコメントの形態素数が 80 を越える事例を無視して学習を行った.

(7) コメント言語モデル学習　(5) で得られた抽象化済み形態素列から日本語コメント文の言語モデルを学習する. 言語モデルは翻訳時の素性の一つとして使われ, よりコメント文らしい出力となるよう翻訳モデルを補正する効果がある. 学習には n-gram 言語モデルのツールキット KenLM[24] を使用し, 5-gram までの情報について言語モデルを構築した.

(8)Tree-to-String 翻訳モデルの学習　以上の手順で得ら

[*2] https://projecteuler.net/

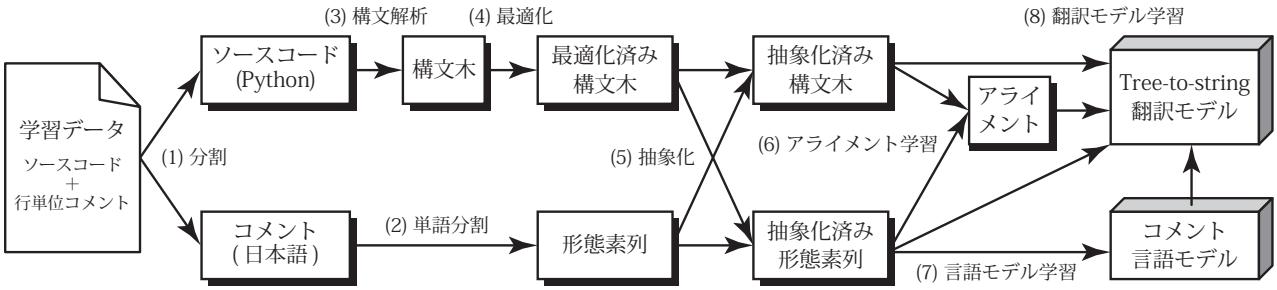


図 7 自動コメント生成のための Tree-to-String 翻訳器の構築

れたソースコード構文木, コメント形態素列, アライメント, 言語モデルを入力とし, Tree-to-String 翻訳モデルの学習を行った. 学習には GHKM アルゴリズム [16] による翻訳ルール抽出と対数線形モデルによるモデル化を行う Travatar[25] を使用した. 重みの最適化に関しては本研究では実施していない.

4.3 比較対象とする提案法の種別

提案法である構文木の最適化と抽象化は, その組み合わせによりいくつかのバリエーションが考えられる. 本研究では以下の条件について翻訳精度の比較を行った.

raw 提案法の最適化, 抽象化を全て省略し, Python の `ast` ライブラリの出力を句構造に成形したものをそのまま使用.

head 最適化法のうち後置型主辞の追加のみを行い, 抽象化は省略.

reduced 後置型主辞の追加, ルールによる木構造の簡略化の両最適化をを行い, 抽象化は省略.

constrained 両最適化に加えて識別子, 定数の抽象化を行う. ただし, 正解コメントを用いた抽象化はテスト時には不可能であるため, テスト時の構文木に現れる識別子と定数は全て抽象化するものとする.

4.4 生成された文の評価

本稿の生成対象であるコメントは自然言語であるため, 各提案法により生成されたコメントは通常の機械翻訳の評価手法に基づいて評価を行った. 機械翻訳の評価法には数式による自動評価と人手による主観評価が存在するが, 本稿では両方の評価を行った.

自動評価に用いる評価尺度には BLEU[26] 及び RIBES[27] を使用した. いずれも n-gram の一致率に基づく尺度であるが, BLEU はより長いフレーズの一致が重視され, RIBES は単語の出現順序の類似性が重視されるという特徴がある. また, いずれの尺度も最低値は 0, 文の完全一致で 1 となるよう設計されている.

主観評価尺度には Acceptability[28] を使用した. これは表 3 に示す 5 段階の評価尺度であり, 評価者にとって翻訳結果がどの程度受け入れられるかを示す指標である.

表 3 Acceptability の定義

評価値	基準
5 (AA)	文法的誤りがなく, 流暢な文.
4 (A)	文法的誤りはないが, 使用単語が不自然.
3 (B)	文法的誤りはあるが, 意味の理解は容易.
2 (C)	文法的誤りがあり, 意味の理解に時間がかかる.
1 (F)	重要な情報が欠落しており, 理解不能.

表 4 各提案法の自動評価尺度

提案法	BLEU %	RIBES %
raw	54.94	86.94
head	60.00	88.96
reduced	63.69	89.77
constrained	56.92	87.46

表 5 各提案法の平均 Acceptability

提案法	平均 Acceptability
raw	3.812
head	4.039
reduced	4.155
constrained	4.174

5. 実験結果

5.1 自動評価

表 4 に, 各コメント生成手法により生成された日本語文の自動評価尺度による評価結果を示す. 使用するコーパスの内, 学習に 9 割, テストに残り 1 割を使用する 10 分割交差検証を行い, 自動評価尺度の値は交差検証で得られる 10 個のテストケースの平均とした.

BLEU, RIBES ともに reduced (主辞の追加+簡略化) が最も翻訳精度が高く, これに head (主辞の追加), raw (最適化なし) が続く. この結果から, 構文木の最適化がコメント生成の精度向上に有効であると考えられる. ところが, constrained(主辞の追加+簡略化+抽象化) に関しては適用前に対して自動評価尺度が低下している. これは全ての識別子と定数を抽象化してしまったため, 抽象化されるべきでない識別子や定数が持っていた文脈情報が失われてしまったためであると考えられる.

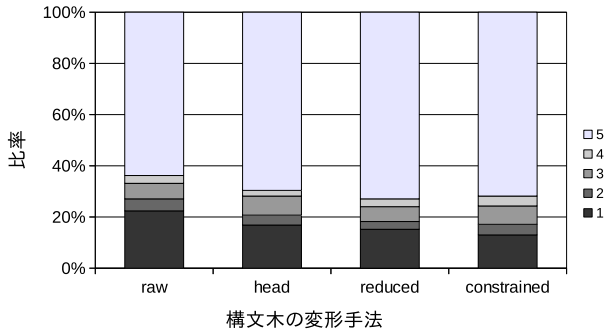


図 8　各提案法の Acceptability 分布

5.2　主観評価

表 5 に各提案法における Acceptability の平均値を示す. また図 8 に, 各提案法における Acceptability の分布を示す.

Acceptability の観点では, いずれの手法でもある程度評価が高く, reduced と constrained においては 70%以上の行に対して正しく, かつ流暢なコメント文を生成できたことが分かる. 最適化を施さない raw においても一定の精度を実現しているのは, Tree-to-String 翻訳では構文木の非終端記号に関する情報を考慮して翻訳ルールを構築するため, 終端記号に関するノイズをある程度吸収しているものと考えられる. また, reduced, head, raw に関して, 構文木の最適化を強く行う手法がより高い評価を得ている点は自動評価尺度と同様である. 一方, constrained の人手評価値が自動評価とは異なり reduced に近い評価値となっている点は注目すべきである. これは constrained によって生成されたコメントの意味は正しいものの, 正解コメントと異なる形式の文が出力され, 文の類似性を調べる自動評価尺度では評価できなかったものと考えられる.

図 8 からは, Acceptability として 5(構文が正しく流暢な文) か 1(理解不能) のどちらかに評価されやすいことが読み取れる. これはソースコード内に現れる識別子や定数は基本的に言い換えることができないことや, ソースコードの示す処理にほぼ曖昧性が存在しないといった特徴を反映していると考えることができ, 生成された日本語文に含まれる語の誤りや係り受けの誤りが許容されないためであると考えられる.

6.　考察

6.1　生成されたコメントの例

表 6 に, 表では 2 種類のソースコード「`if x % 5 == 0:`」「`if x % 5 == 1:`」に対して各提案法により生成されたコメントを示す. 両ソースコードはいずれも剰余を求めて定数と比較しているものだが, 比較対象の定数が 1 のときは「割った余りが 1」, 0 のときは「割り切れるなら」といった翻訳をされていることが分かる. これは剰余の文脈で定数 0 と比較している場合に「割り切れる」という解釈がされることを反映していると考えられる. このように特定の文脈に関するコメントを生成する場合, 従来法のようなルールに基づく手法では, 新たな規則を発見する度に人手でルールを更新しなければ対応するのが難しいが, Tree-to-String 翻訳ではそのような特徴を含むデータを収集して学習を行えばよく, システムの更新が容易となる. これは提案法の利点の一つである.

constrained の場合については, 生成結果が両方とも「割った余りが · であれば」となってしまっている. これはソースコードに含まれる全ての識別子と定数が抽象化されてしまったことで, 特定の値に関するルールを適用できなかったためと考えられる. この場合生成されたコメント文自体に間違いはないため主観評価は高くなるが, 正解コメントとの類似度が低下するため, 自動評価尺度は低く産出されてしまうと考えられる. これは表 4, 表 5 のような実験結果が得られた直接の原因と考えられる.

constrained においても正解コメントに近いコメントの生成を行うためには, ソースコードに含まれる全ての識別子と定数について抽象化する場合としない場合で翻訳を行い, 最も尤度の高い翻訳結果を採用する手法が考えられる. これは単純な Tree-to-String 翻訳で行おうとすると識別子と定数の出現個数の指数関数に比例するだけの構文木を翻訳しなければならないが, 組合せの表現を工夫することで原理的に Tree-to-String 翻訳と同じ計算量で処理可能であることが知られている [12].

7.　おわりに・今後の課題

本稿では Tree-to-String 翻訳の枠組みを用いて, ソースコードから得られる構文木とソースコードに付与された文単位のコメントの対からコメントを自動的に生成する手法を提案した. この手法では, ソースコード上の文脈を人手でルール化する従来法に比べてソースコードとコメントの対さえあればコメント生成器を学習できる利点がある.

今後の課題としては, ソースコード 1 文単位のコメント生成ではなく, より大きな単位を用いたコメント生成を行うことが考えられる. ソースコードは通常複数の文が集まって一つのアルゴリズムを構成しており, 文単位のコメント生成ではアルゴリズム名などの全体的な処理を説明するコメントに対して正しく学習することができないと考えられる. これに対しては, 複数の文, または制御単位ごとのコメント生成を行うことが考えられる.

また, 本研究では構文木の最適化に人手で定義された変換ルールを用いたが, 定義されたルールは Python のみに適用できるものであり汎用的ではない. 対象とするプログラミング言語ごとに自動で変換ルールを構築することができれば, プログラミング言語に非依存な自動コメント生成器を構築することができる. これに関して, 機械翻訳の分野では人手によるルールではなく対訳データの整合性に基

Python	`if x % 5 == 0:`	`if x % 5 == 1:`
raw	もし x が 5 で 割っ た 余り	x が 5 で 割り切れる 1 と 等しけれ ば
head	x 5 で 割り切れる なら	x 5 で 割っ た 余り が 1 で あれ ば
reduced	もし x が 5 で 割り切れ る なら	もし x を 5 で 割っ た 余り が 1 で あれ ば
constrained	もし x を 5 で 割っ た 余り が 0 で あれ ば	もし x を 5 で 割っ た 余り が 1 で あれ ば

表 6 生成されたコメントの例

づいて変換ルールを自動的に構築する手法が提案されており [20], 本手法への適用も考えられる.

今回使用したデータはソースコードの記述者とコメントのアノテータが少数であり, 統計的機械翻訳の学習データとしては小規模である. 今後のデータ収集の方針として, 既に大量のプログラムの蓄積があるオープンソースプロジェクトからソースコードを収集し, これに対してコメントのアノテーションを行うことを予定している.

また, 本手法によって生成されたコメントをソースコードと共に学習者へ提示することで, 実際の読解速度や理解度にどのように影響を及ぼすかについても実験, 検証を行う.

謝辞

本研究の一部は, 頭脳循環を加速する戦略的国際研究ネットワーク推進プログラムの助成を受け実施したものである.

参考文献

[1] DeLine, R., Venolia, G. and Rowan, K.: Software Development with Code Maps, *Commun. ACM*, Vol. 53, No. 8, pp. 48–54 (2010).
[2] Rahman, M. M. and Roy, C. K.: SurfClipse: Context-Aware Meta Search in the IDE, *Proc. ICSME*, pp. 617–620 (2014).
[3] Sridhara, G., Hill, E., Muppaneni, D., Pollock, L. and Vijay-Shanker, K.: Towards automatically generating summary comments for Java methods, *Proc. ASE*, pp. 43–52 (2010).
[4] Buse, R. P. and Weimer, W. R.: Automatic Documentation Inference for Exceptions, *Proc. ISSTA*, pp. 273–282 (2008).
[5] Sridhara, G., Pollock, L. and Vijay-Shanker, K.: Automatically Detecting and Describing High Level Actions Within Methods, *Proc. ICSE*, pp. 101–110 (2011).
[6] Wong, E., Yang, J. and Tan, L.: AutoComment: Mining question and answer sites for automatic comment generation, *Proc. ASE*, pp. 562–567 (2013).
[7] Brown, P. F., Pietra, V. J. D., Pietra, S. A. D. and Mercer, R. L.: The mathematics of statistical machine translation: Parameter estimation, *Computational Linguistics*, Vol. 19, No. 2, pp. 263–311 (1993).
[8] Koehn, P., Och, F. J. and Marcu, D.: Statistical phrase-based translation, *Proc. NAACL-HLT*, pp. 48–54 (2003).
[9] Chiang, D.: Hierarchical phrase-based translation, *Computational Linguistics*, Vol. 33, No. 2, pp. 201–228 (2007).
[10] Huang, L., Knight, K. and Joshi, A.: Statistical syntax-directed translation with extended domain of locality, *Proc. AMTA*, Vol. 2006, pp. 223–226 (2006).
[11] Neubig, G. and Duh, K.: On the Elements of an Accurate Tree-to-String Machine Translation System, *Proc. ACL*, Baltimore, USA (2014).
[12] Mi, H., Huang, L. and Liu, Q.: Forest-Based Translation, *Proc. ACL-HLT*, Columbus, Ohio, pp. 192–199 (2008).
[13] Brown, P. F., Pietra, V. J. D., Pietra, S. A. D. and Mercer, R. L.: The Mathematics of Statistical Machine Translation: Parameter Estimation, *Computational Linguistics*, Vol. 19, No. 2, pp. 263–311 (1993).
[14] Och, F. J. and Ney, H.: A Systematic Comparison of Various Statistical Alignment Models, *Computational Linguistics*, Vol. 29, No. 1, pp. 19–51 (2003).
[15] Neubig, G., Watanabe, T., Sumita, E., Mori, S. and Kawahara, T.: An Unsupervised Model for Joint Phrase Alignment and Extraction, *Proc. ACL-HLT*, Portland, Oregon, USA, pp. 632–641 (2011).
[16] Galley, M., Hopkins, M., Knight, K. and Marcu, D.: What's in a Translation Rule?, *Proc. NAACL-HLT*, pp. 273–280 (2004).
[17] Och, F. J.: Minimum Error Rate Training in Statistical Machine Translation, *Proc. ACL*, Sapporo, Japan, pp. 160–167 (2003).
[18] Isozaki, H., Sudoh, K., Tsukada, H. and Duh, K.: Head Finalization: A Simple Reordering Rule for SOV Languages, *Proc. WMT*, Uppsala, Sweden, pp. 244–251 (2010).
[19] Hatakoshi, Y., Neubig, G., Sakti, S., Toda, T. and Nakamura, S.: Rule-based Syntactic Preprocessing for Syntax-based Machine Translation, *Proc. SSST*, Doha, Qatar, pp. 34–42 (2014).
[20] Burkett, D. and Klein, D.: Transforming Trees to Improve Syntactic Convergence, *Proc. EMNLP*, Jeju Island, South Korea (2012).
[21] Li, P., Liu, Y. and Sun, M.: An Extended GHKM Algorithm for Inducing λ-SCFG, *Proc. AAAI* (2013).
[22] 小田悠介，ニュービッグ・グラム，サクティ・サクリアニ，戸田智基，　中村哲：自動プログラミングへ向けた問題解答コーパスの収集と考察，情報処理学会自然言語処理研究会報告，Vol. 2014, No. 22, pp. 1–8 (2014).
[23] Kudo, T., Yamamoto, K. and Matsumoto, Y.: Applying Conditional Random Fields to Japanese Morphological Analysis., *Proc. EMNLP*, Vol. 4, pp. 230–237 (2004).
[24] Heafield, K., Pouzyrevsky, I., Clark, J. H. and Koehn, P.: Scalable Modified Kneser-Ney Language Model Estimation, *Proc. ACL*, Sofia, Bulgaria, pp. 690–696 (2013).
[25] Neubig, G.: Travatar: A Forest-to-String Machine Translation Engine based on Tree Transducers, *Proc. ACL*, Sofia, Bulgaria, pp. 91–96 (2013).
[26] Papineni, K., Roukos, S., Ward, T. and Zhu, W.-J.: BLEU: A Method for Automatic Evaluation of Machine Translation, *Proc. ACL*, pp. 311–318 (2002).
[27] Isozaki, H., Hirao, T., Duh, K., Sudoh, K. and Tsukada, H.: Automatic Evaluation of Translation Quality for Distant Language Pairs, *Proc. EMNLP*, pp. 944–952 (2010).
[28] Goto, I., Chow, K. P., Lu, B., Sumita, E. and Tsou, B. K.: Overview of the patent machine translation task at the NTCIR-10 workshop, *NTCIR-10* (2013).