D-048

Hadoopを用いた大規模日本語コーパスにおける可変長単語N-gram頻度計算手法

A variable-length word N-gram frequency calculation method in large-scale Japanese corpus using Hadoop

小笹 哲哉 [‡] **渋木 英潔** [†] **森 辰則** [†]
Tetsuya KOSASA Hideyuki SHIBUKI Tatsunori MORI

1 はじめに

自然言語処理において、単語N-gram頻度は単語の共起関係を調べる他に、単語N-gram言語モデルを用いる仮名漢字変換 [1] や音声認識 [2]、機械翻訳 [3] などといった応用技術に使われている。単語N-gram言語モデルは、直前の $(N-1)$ 個の単語列を状態とした $(N-1)$ 次のマルコフモデルにより、次の単語の条件付き確率を計算していくといった、確率的言語モデルの一つであり、文の出現しやすさを数学的なモデルで表したものである。日本語においては、単語の境界が曖昧であり、短い助詞の連続などによって長い N が頻出してしまう場合があるため、短い N で固定するより、文脈に応じて必要な長さを用いるような可変長単語N-gramモデルが求められている。実際に提案されている可変長単語N-gramモデルでは、N が最大、すなわち最も長い文の単語数となるN-gramをまず最初に作る必要がある [4]。ここで計算対象となる単語の組み合わせは N が大きくなるほど指数的に増えていく。また、単語N-gram頻度を計算する対象、例えば、日本全国の地方議会録などといった言語コーパスは年々大規模化しており、限られた計算資源において扱う場合、主記憶容量が足りないことや、計算時間が莫大になるなどといった問題がある。そのため、大規模な言語コーパスに対して、特に N が大きい場合の単語N-gram頻度を計算する際に効率的な手法が求められている。

N の大きさが決められている固定長の単語N-gramは、大規模な文章に対しては近似的に計算していく手法 [5] が主流となっているが、N を最大のオーダーまで計算するような可変長単語N-gramの頻度を計算する方法としては、Suffix Arrayを用いて行う手法 [6] が提案されている。しかし、この手法では、Suffix Arrayを大規模な文章に対して構築することについては述べられておらず、Suffix Arrayを構築する手法について考える必要がある。単純にSuffix Arrayを構築する手法では文字列の長さを n とした計算量が $O(n^2 \log n)$ となっており [7]、本稿にて扱うような非常に大きな n に対しては非効率的であると言える。このため、まず大規模な文章に対してSuffix Arrayを構築することが必要である。

そこで、本稿では大規模日本語コーパスに対して可変長単語N-gram頻度の計算を行う手法として、主記憶容量に収まらないような大規模なデータを扱うため、処理を分散させることを考えた。そのため、本稿では、効率的に分散処理することが可能となるHadoopを用いてSuffix Arrayを並列分散して構築し、可変長単語N-gram頻度を計算することを目的とする。Hadoop[8][9] を用いるためにはMapReduceモデルでアルゴリズムを設計しなければならないため、本稿では、Suffix Arrayの構築をMapReduceのモデルに則って並列分散化する。そのために、並列化してSuffix Treeを構築できるMansourら [10] のアルゴリズムを利用することを考えた。しかし、彼らはDNA配列を対象にアルゴリズムを設計しており、本稿で扱う日本語の単語に対しアルゴリズムをそのまま適用すると処理が低速になってしまう問題がある。これについて、低速な処理を並列分散化することで解決を試みた。つまり、本稿で提案する手法では、Mansourら [10] の手法について、MapReduceのモデルに則ってアルゴリズムを設計しなおす。MapReduceモデルに適用する際に、後に述べるように1)Mapによる分割が並列化できない、2)Reduceによる集約の活用がされない、3) 処理を並列化する単位の見積もりにおいて、元文章を数多く走査しなければならず、ファイル入出力が多い、という問題がある。これらについて、1) サンプリングによって語の頻度を近似計算することで並列化を行う、2)Mapにおいて先頭語の位置情報を出力する、3) 分割する語の長さを事前に与えることでファイル入出力を削減する、とすることにより解決を試みた。

最後に、本論文の構成について説明する。第2章ではSuffix Arrayを構築するための方法として、Suffix Treeの並列構築法についての詳細を述べる。第3章では並列分散するために用いるHadoopの詳細について述べる。第4章ではHadoopを用いることによってどのように並列分散化を行うかについて検討し、第5章で、第4章での議論を踏まえて、分散処理によってSuffix Arrayを構築し、可変長単語N-gram頻度を求める手法について説明する。第6章では、提案手法の評価実験を行い、その結果を考察する。最後に第7章で結論を述べる。

2 Suffix Treeの並列構築法

可変長のN-gram頻度を求める研究としては、Nagaoらによる方法 [11] やYamamotoら [6] による方法がある。Yamamotoら [6] はSuffix Arrayを用いて任意の長さに対して、N-gramを求める方法を提案した。

しかし、長さが n である文字列 T についてSuffix Arrayを構築する場合、全体の計算量は $O(n^2 \log n)$ となる [7] ことから、入力文章が大規模になるとSuffix Arrayの構築が難しい。そのため、事前に大規模なSuffix Arrayについて構築する方法について考える必要がある。ここで、

[†]横浜国立大学大学院環境情報研究院, Graduate School of Environment and Information Sciences, Yokohama National University
[‡]横浜国立大学大学院環境情報学府, (ditto)

Copyright © 2014 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.

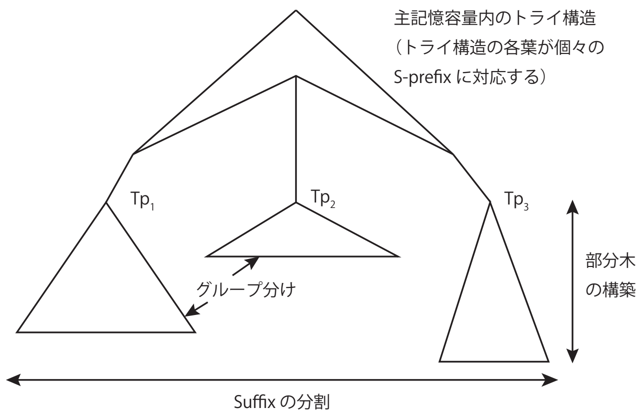

図 1: ERA アルゴリズムでの Suffix Tree 分割

Suffix Tree を構築することができれば Suffix Array を構築することができることが知られているので、大規模な文章について Suffix Tree を構築することを検討する。

2.1 Suffix Tree 構築法の概要

Suffix Tree を構築する手法として、Mansour ら [10] は大規模な文字列に対して効率良く構築できる ERA アルゴリズムを提案している。ERA アルゴリズムでは構築する Suffix Tree を主記憶容量に収めることを考え、Suffix Tree を構成する Suffix について分割を行い、その Suffix に対して部分木構造を構築する。ERA アルゴリズムでの Suffix Tree 構築においてどのように部分木に分割するか示した図を図 1 に示す。

まず、部分木に分割するため、部分木の先頭部分に対応する S-prefix に応じて Suffix の集合を分割し、グループ化を行う。なお、ある文字列 S についての Suffix の先頭単語列の各々を S-prefix と呼ぶ。次に、決定した S-prefix から部分木を構築しマージすることで全体の Suffix Tree を構築している。

2.2 入力文の Suffix への分割

ERA アルゴリズムでは Suffix Tree を構築する際に、まず主記憶容量に収まるように対象文章に対応する全 Suffix からなる集合を、同じ S-prefix で始まるものに分類して部分集合に分割する。これを、ERA アルゴリズムでは VerticalPartitionig と名づけている。

ある S-prefix から始まる文字列が N 個存在する場合、その S-prefix を根とする部分木の葉の数は N となることから、文章中に現れる文字の頻度を計算する。

ここで、部分木の大きさが主記憶容量に収まらない場合は、部分木が主記憶容量に収まる大きさになるまで、S-prefix の長さを増やしていけば良い。これは S-prefix の長さが長いほど文章内での頻度が小さくなる、すなわち、部分木の大きさが小さくなるためである。S-prefix の長さを増やして調整する操作を文章内に出現するすべての S-prefix に対して行い、各 S-prefix で始まる Suffix が主記憶容量に収まるようにする。

最後に、主記憶容量を最大まで利用するため、グループ分けを行う。これは、先ほど分割した S-prefix を別々のグループとして割り当てた時、頻度が少ない S-prefix がある場合、主記憶容量を活用できないため、複数の S-prefix を一つのグループとして割り当てることを指す。グループ内にある全ての S-prefix の頻度の合計が、主記憶容量に収まる大きさになるようにまとめ上げる作業を行う。

なお、本稿では S-prefix として単語を扱う。Mansour ら [10] では文字で扱っているが、単語を一つの単位としても文字列の比較、頻度の計算についても、文字の場合と同様の方法で行うことができる。

2.3 部分 Suffix Tree の構築

ERA アルゴリズムではある S-prefix を先頭にもつ Suffix に対し、部分 Suffix Tree を構築する。これを ERA アルゴリズムでは SubTreePrepare と名づけている。

まず、第 2.2 節で説明した方法により調整された文字列 (S-prefix) を入力とし、この文字列が入力文章のどこに出現するのかを全て枚挙する。

当該文字列を S-prefix に持つ部分 Suffix Tree を構築するためには、全ての枝情報の決定が必要となる。つまり、木を構成する葉がすべて辞書式順序となっている必要がある。これは、枚挙した全ての位置から始まる文字列を辞書式順序にソートすることに相当する。このため、出現位置の次の位置から始まる文字列 (Suffix) を配列に読み込み、辞書式順序にソートすることを行う。

しかし、大規模な文章を対象とする場合、文章の最後まで文字列を読み込むと主記憶容量に収まらない場合が発生する場合がある。そのため、読み込む文字列の長さは、すべての配列に文字列を読み込んだ時に、主記憶容量に収まる一定の長さに制限する必要がある。

ここで、読み込んだ文字列の長さが短い場合を考えると、読み込まれている範囲では同じ文字列に見えても、それよりも後方で文字列が異なることがあるので、その配列に対してソートを行っても、すべての配列の要素が辞書式順序になっているとは限らない。そのため、読み込まれている範囲において同じ文字列となっている Suffix 集合に対しては、読み出す文字列の範囲を後方にずらし、再度文字列を一定の長さだけ読み取り、辞書式順序になるまで繰り返し行う。また、辞書式順序となっている要素に対しては Suffix Tree の枝情報を決定することができる。

部分 Suffix Tree の枝情報が全て決定され、全ての要素が辞書式順序となった時、部分 Suffix Tree が完成していることになる。

3 Hadoop による分散処理

Hadoop[8][9] は Apache Software Foundation により開発, 運用されているオープンソースソフトウェアの一つである。Hadoop を用いると、MapReduce モデルにより、大規模データを効率的に分散処理、および管理することができる。

この章では第 2 章で説明した Suffix Tree 構築法を Hadoop 上で扱うための技術について述べる。

3.1 MapReduce とジョブ

MapReduce は、データ処理のためのプログラミングモデルである。MapReduce においてクライアントが実行を要求する作業単位をジョブという。ジョブは、入力

Copyright © 2014 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.

データ、MapReduce のプログラム、設定情報から構成される。Hadoop はジョブをタスクに分割して実行する。タスクには map タスクと reduce タスクという 2 種類がある。map タスクの中では入力を分割して出力する役割を持つ Mapper が、reduce タスクの中では入力を集約して出力する Reducer が内部で実行される。

ジョブへの入力は、入力スプリットと呼ばれる固定長の断片に分割され、各スプリットに対して 1 つの map タスクが生成される。map タスクではユーザが設定した Mapper クラス内の map 関数を、スプリット中の各レコードに対して実行する。map タスクの出力は reduce タスクが実行されているノード上へコピーされ、一つの入力としてマージされたものがユーザが設定した Reducer クラス内の reduce 関数に渡される。

map および reduce 関数は、以下のように入力と出力にそれぞれキーと値を持つ。

map: (K1, V1) → list(K2, V2)
reduce: (K2, list(V2)) → list(K3, V3)

Hadoop 上で分散処理を動作させるにはこのような MapReduce のプログラミングモデルに沿って処理を記述する必要がある。

3.2 シャッフルとソート

map タスクの出力はソートされ、ソートされた map タスクの出力は reduce タスクが実行される計算ノードへネットワークを通じてコピーされる。この動作をシャッフルという。複数の Reducer がある場合、map タスクはその出力をパーティション化し、それぞれの reduce タスクに対して 1 つのパーティションを生成する。同じキーの対するレコードは単一のパーティションに格納される。

しかし、複数の reduce タスクがある場合、各 reduce タスクの出力は Reducer 内ではソートされるが、Reducer 間を含めた全体としての順序は保証されない。

3.3 全体ソートにおけるサンプリング

Partitioner は、Hadoop において複数の reduce タスクがある場合、どの reduce タスクへ渡すかどうかを決定する役割を担っている。Hadoop を利用して、全体としてソートされたファイルを出力することを考えた時、出力の全体としての順序を尊重するような Partitioner を利用する必要がある。

map 関数の入力のキーの集合に対して、ある一定の数値を固定してパーティション分けを行うと、特定のパーティションに値が偏ることが想定される。しかし、Partitioner はジョブを実行する際に、map 関数や reduce 関数同様、複数の処理が実行される計算ノード上にコピーされてから実行される。そのため、どの reduce タスクに割り振るかどうかという情報は、どの計算ノードからでも同じ情報が事前に共有されている必要がある。つまり、均等なパーティションの集合にするためには、入力のキーの空間の分布を推定する必要がある。本稿では、入力のキーをサンプリングし、キーの分散状況を近似的に推定する。この情報は、Hadoop 上のファイルシステム、あるいは Hadoop 上の分散キャッシュに追加しておくことで、どの計算ノードからでも参照できるようにする。

3.4 入力フォーマット

Hadoop では、多様なデータ形式に対して処理を行うことができる。本節では Hadoop が利用できる形式について述べる。

入力スプリットは入力の断片であり、1 つの map によって処理される。各 map は 1 つのスプリットを処理する。各スプリットはレコード、つまりキーと値のペアに分割され、map はそれらのレコードを順番に処理する。多くの場合、各スプリットとレコードはファイルに結び付けられている。例えば、本研究でも扱う TextInputFormat では、Hadoop 上のファイルシステムにある文章を、ある大きさの入力スプリットに分割し、キーをファイルの先頭バイトオフセット、値をキーに該当する行とするようなレコードに分割される。ここでのスプリットの大きさは通常、Hadoop 上のファイルシステムのブロックサイズの大きさとなっている。

1 つの mapper が入力ファイル全体を処理させるには、ファイルを分割させないようにする必要がある。既存のファイルが分割されないようにする方法としては、最小スプリットサイズをファイルサイズより大きくする方法と、使いたい InputFormat クラスを継承して分割しないようにする方法がある。

3.5 複数処理の連結

MapReduce モデルでは、1 つのジョブにおいて、キーと値を用いて入力データ、map 関数、reduce 関数、出力データと繋げている。しかし、実際の Hadoop アプリケーションでは、1 つ以上の MapReduce ジョブからなる。したがって、抽象化して考えた場合、キーと値は map と reduce に結びつけるだけではなく、reduce から次の map、などといった結びつけを行うことになる。

このように複数の map や reduce の結びつけを行う API として Cascading[12] といったものがある。Cascading はキーと値を抽象化し、対応するフィールド名を持つタプルというものに置き換えることによって複数の map や reduce の結びつけを簡単に行えるようにしている。

しかし、Cascading などの API を用いても、外から 1 つのジョブを繰り返し行うような処理を記述することは出来ない。そのため、ループ構造を含む記述を行う場合はその処理を map 関数、あるいは reduce 関数の中に内包する構造にするか、ループ構造そのものを無くすようなアルゴリズムを変更する必要がある。

4 Hadoop を用いた Suffix Array 構築の並列分散化に関する検討

本章では第 2 章で説明した手法を Hadoop を用いて分散処理する方法について検討する。

4.1 基本的な考え方

第 1 章で説明したように、可変長単語 N-gram 頻度を求めるためには、最大オーダーの N まで頻度を求める必要がある。これを行うには、Yamamoto ら [6] の手法を用いれば良い。Yamamoto らの手法で頻度を求めるには Suffix Array が必要となるので、本稿で対象とするような大規模な文章で Suffix Array を構築することについて考えてみる。Suffix Array は文字列中のすべての Suffix について文字列比較を行い、辞書式順序でソートを行う

Copyright © 2014 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.

ことで構築することができる。ここで、文字列の長さを n とすると、$O(n^2 \log n)$ という計算量になる。そのため、文字列の長さが長くなるにつれ、構築にかかる時間が長くなる。また、文字列をソートする際に文字列を主記憶装置へと読み込むが、主記憶容量に収まる大きさでなくなることが考えられる。

ここで、Suffix Array は Suffix Tree の葉を辞書式順序に辿って行くことでも構築することが可能であるので、Suffix Tree を構築するという問題に置き換えることができる。Suffix Tree を大規模な文章について構築する手法としては、第 2 章で説明した Mansour ら [10] による ERA アルゴリズムが存在する。ERA アルゴリズムでは、構築する Suffix Tree を分割することでメモリ使用量を抑えつつ、並列に Suffix Tree を構築している。しかし、この ERA アルゴリズムでは、単語種類数が多い場合、第 2.2 節で説明したような入力文を Suffix へと分割する際に行う Suffix の文章内に出現する頻度の計算が多くなることから計算する時間が長くなるという問題と、第 2.3 節で説明したような文字列の長さを主記憶容量に収まる長さに制限して文字列を読み込む際に、最終的な順序を決定できなかった場合、元の文章へ再度アクセスして読み込む位置をずらして文字列を読み込み比較するためファイル入出力の回数が増えるといった、効率面での問題が発生する。

そこで、この ERA アルゴリズムを効率的に大規模データを扱うことのできる Hadoop で実現することにする。

4.2 入力文の Suffix 分割における並列分散化

入力文の Suffix 分割の詳細な方法は第 2.2 節で述べた。入力文の Suffix 分割において、出力すべきデータについて考えてみると、主記憶容量に収まるような大きさの Suffix Tree を後に作れるような S-prefix が必要である。そのため、このような S-prefix を切り出すためにどのような処理を MapReduce で記述すれば良いかを考えてみる。ここで一番計算するのに時間がかかる処理は、文章内のある S-prefix の頻度を求めることである。S-prefix が主記憶容量に収まる大きさの木を生成するかどうかを調べるために頻度を計算する際にファイル入出力が発生する。これを文章に含まれるすべての S-prefix に対して実行するため処理に時間がかかる。つまり、この部分を分散化することができれば、処理全体の高速化につながることが考えられる。

まず、ある長さ N の全ての S-prefix の頻度を計算するという処理について、以下の様なジョブを定義し、行うことを考えてみた。

map 関数では入力で対象文章を一行ごと読み込み、S-prefix の長さと同じ単語列を生成し、そして、それぞれの単語列について頻度数 1 という情報を出力する。つまり、map 関数では S-prefix と頻度 1 という情報が出力される。reduce 関数では入力でキーを S-prefix、値を S-prefix の頻度数の情報を受け取り、入力キーとその値の合計という情報を出力することで、全ての長さ S-prefix について頻度を求めるという処理を実現することができる。

しかし、このジョブの計算結果である頻度から主記憶容量より大きい大きさの木が生成される場合は繰り返しこのジョブを実行する必要がある。そのため、このジョブを用いて 1 つの単語を計算するという処理を行う場合は、ジョブを繰り返し実行するという処理を記述する必要がある。しかし、ジョブの繰り返し構造を記述することは Hadoop の枠組みでは困難であるため、このようなジョブを定義しない方法として、map 関数で繰り返しを内包して指定頻度となるまで S-prefix の長さを加えていくことを考える。この場合は Mansour ら [10] の Vertical Partitioning アルゴリズムがそのまま map 関数となる。MapReduce のプログラミングモデルにおいては、各 Mapper が実行している自身以外の Mapper の情報を共有することができないことから、Mapper 数を 1 つに限定して、Vertical Partitioning を実行することになる。しかし、この場合、分散することができておらず、Hadoop で実行することによる恩恵をあまり受けることができないと考えられる。

そのため、Mapper の数を複数に増やすことができるような方法を考える必要がある。ここで、出力すべきデータは主記憶容量に収まるような長さの S-prefix の集合であった。つまり、事前に主記憶容量に収まるような S-prefix となるように単語をどの程度の長さになるまで連結すればよいかを与えれば、Mapper では S-prefix に切り出すというタスクのみを実行するように考えれば良いことになる。また、切りだした S-prefix を全体の結果において辞書式順序でソートするためには、どの Reducer へ割り当てると良いかを決定するグループ分けについては、Mapper に入力に対してサンプリングを行うことを考える。つまり、サンプリングした S-prefix の集合から全体の集合の近似を得ることにより、ジョブの実行結果全体として辞書式順序でソートされつつ、各 Reducer に割り当てるすべての S-prefix の合計頻度が均等になるように分散することが可能となると考えられる。

4.3 部分 Suffix Array の構築の並列分散化

部分 Suffix Tree 構築の方法は第 2.3 節で述べた。Suffix Array を構築する場合は Suffix Tree の葉の情報のみを出力すれば良いので、ここでは Suffix Array の情報について出力できれば良いことになる。ここでも時間のかかる処理はファイルの入出力であるので、入力文章の読み込みが発生する箇所について並列分散化することを考える。

部分 Suffix Tree を構築するためには、構成する葉はすべて辞書式順序でソートされている必要があるので、ERA アルゴリズムでは配列に一定数の単語をファイルから読み込み、ソートを行っていた。

ここで、配列に一定数の単語を格納する際のファイル入出力について考える。この場合、単語を格納する時点での文字列読み込みだけでなく、その後のソートと枝情報の更新を一つのジョブとして実現できるかを考えてみる必要がある。つまり、ジョブの繰り返しを発生させずに 1 回の処理で枝情報をすべて決定するということを行う必要がある。このような処理を行うジョブを、map 関数では文字列を読み取り、出力を全体の順序を保証する Partitioner を利用して、reduce 関数で枝情報を決定することとしてみる。文字列の読み取りについては、単語の位置情報、読み取る開始位置と読み取る単語数が分かれば該当する文字列を出力できる。ここでは、map 関数

Copyright © 2014 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.

を入力のキーについては入力単語の位置情報、値を読み取る開始位置と単語数のペアとし、出力のキーを該当する文字列、値を入力単語の位置情報とする。次に、ソートについてはキーの単語に対し、辞書式順序でソートさせる必要がある。ここではキーについて全体としての順序を保証させるような Partitioner を実装することにする。最後に、reduce 関数は入力のキーを文字列、値を入力単語の位置情報として考えると、キーについて辞書式順序になって出力されていることが保証されていれば良いことになる。Hadoop では各 Reducer の出力についてはキーによってソートされるので、reduce 関数では何も処理を行わなくても、Hadoop 側がソートしてくれることになる。しかし、一度で枝情報が決定するには、入力文章によっては、非常に長い文字列を map 関数で出力する必要があるため、あまり効率が良い方法とは考えられない。

そのため、配列に一定数の単語を格納する際のファイル入出力については繰り返しを排除して一度で決定するような処理にはしないほうが良いと考えられる。つまり、ERA アルゴリズムの部分 Suffix Tree 構築のアルゴリズムを Reduce 関数で内包するような方法を取れば良いと考えられる。

また、ファイル入出力が発生する箇所として、単語についての出現位置も列挙する必要があり、この部分に関しても処理が低速化する要因となっていると考えられるため、この部分についても削減する方法を取るように考えてみる。ここで、reduce 関数は、map 関数の出力のキーについてすべての値をリストとしてまとめることを考えてみると、map 関数にて単語の出現位置を値として出力してやれば、出現位置をすべて列挙するためのファイル入出力を削減できるのではないかと考えられる。

5 提案手法

本章では、可変長な単語 N-gram について頻度を計算するために、分散処理によって Suffix Array を構築する手法について述べる。

5.1 全体の流れ

図 2 に提案手法の概要を示す。本稿では、入力とする日本語の文章に対して、一文ごとに形態素解析器によって空白記号を用いて分かち書きを行う。Hadoop 上では、改行区切りで一行を認識するため、一文を改行で区切る。このように処理した文章を Hadoop 上のファイルシステムに配置する。Hadoop を用いて Suffix Array を分散処理を行い構築する。Hadoop 上で Suffix Array を構築する際に、行うべき処理としては以下のようなものがある。

- Hadoop のファイルシステムから受け取った一文を Suffix へと分割
- 全体として出力が辞書式順序となるようにソート
- 各 Reduce フェーズで Suffix をソートし、部分 Suffix Array を構築

これらの処理を一つのジョブとして Suffix Array を構築する。Hadoop 上で構築された Suffix Array を用いて、第 2 章で説明した Yamamoto ら [6] の手法によって単語 N-gram 頻度を計算する。

5.2 Suffix への分割と Suffix の位置情報枚挙

まず、第 4.2 節で述べたように、主記憶容量に収まる大きさの木を構築するために Suffix へ分割する。提案手法では、これを map 関数で実行することを考えた。

Mansour ら [10] の手法では DNA 配列を入力文章として Suffix Tree を構築していた。しかし、本稿で扱う文章は日本語であり、単語についての異なり数は DNA 配列と比較すると非常に多い。そのため分割する Suffix が 1 単語、つまり、1-gram であったとしても、入力文章が非常に大規模ではなかった場合については主記憶容量に収まる大きさになると考えられる。

そのため、map 関数での入力文については単語 1-gram として切り出して出力してみることにする。

また、第 4.3 節で述べたように、部分木の構築について、単語が入力文章中のどこに出現するのかを決定する際にファイル入出力が発生する。提案手法では、これについても map 関数で行うように改善することができると考えた。

map 関数の入力について、Hadoop の入力フォーマットの 1 つである TextInputFormat を用いることによって、入力のキーにファイルのバイト数オフセット、値にはキーを先頭バイト数とした該当行を割り当てる。次に、単語 1-gram について分割する際、その行の何単語目になるかという数値とともに出力する。つまり、出力については、キーに単語、値にファイルのバイト数オフセットと該当行の何単語目かの情報のペアとして出力する。

これによって、map 関数にて単語が入力文章中のどこに出現するのかについて出力され、reduce 関数の入力にて、同一キーに対して値がまとめられ、文章内のすべての出現位置について枚挙することができる。

5.3 サンプリングされた語の頻度に基づく Suffix の概算ソート

同一の Reducer に渡される Suffix については、順序付けがきちんとされているが、異なる Reducer に渡される Suffix についてはまだされていない状況である。「概算ソート」の意味として、異なる Reducer についてもジョブの実行結果全体から見てソートされるようにすることを言う。

map 関数の出力として単語がキーとなるが、このキーに対して全体としての順序が辞書式順序になっていなければならない。ここで、全体の順序を保証するような Partitioner を用いることを考える。

Partitioner では、あるキーと値が与えられた時に、どの Reducer へ割り当てるかを決定する。そのため、全体の順序を保証するような Partitioner を使う場合、与えられたキーと値があるデータに対して比較した際の大小によって決定する。この比較対象となるデータを決定するために、Partitioner では、入力に対して文字列をサンプリングする。サンプリングした結果より、全体の分散を近似によって求め、データがどの部分で区切ればよいかを決定する。

今回の場合は Mapper の出力である Suffix について辞書式順序になっていれば良い。そのため、本提案手法ではサンプリングは単語に対して行うようにした。また、サンプリングについてはランダムに入力文章から一

Copyright © 2014 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.

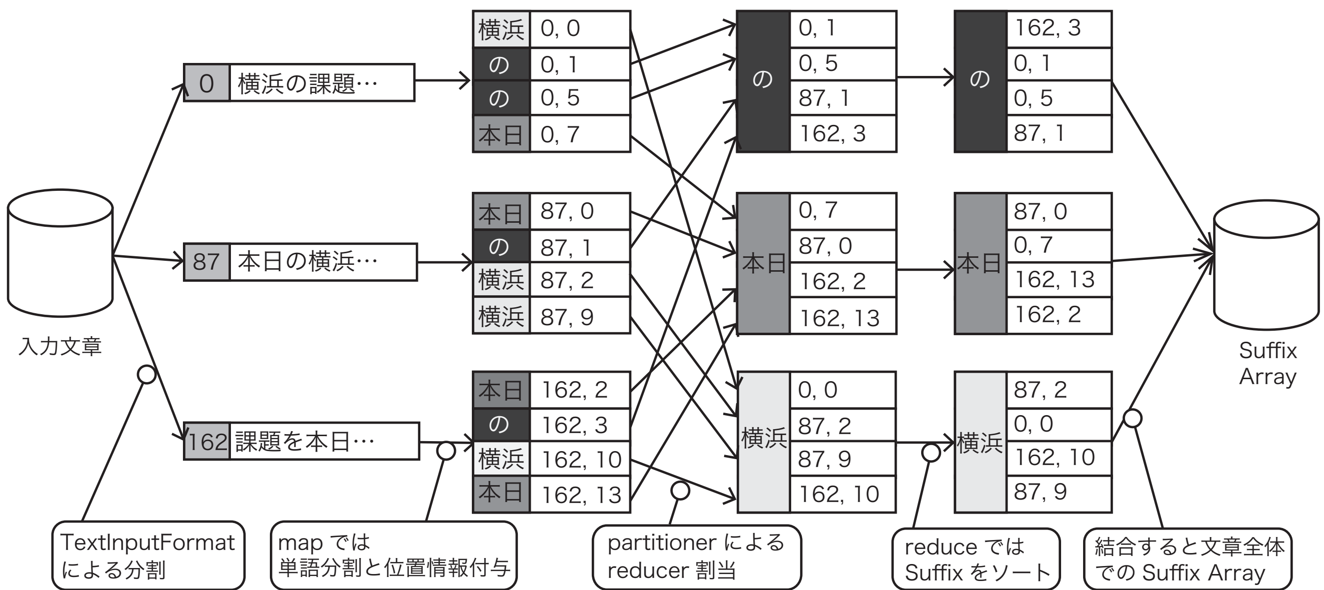

図 2: 提案手法の概要

行を取り出すという処理を一定数行い、それらに対して単語を切り分けて行うことにした。また、サンプリングされた単語の頻度がわかるので、それを S-prefix に持つ Suffix の頻度が相対的にわかったことになる。これによって Suffix をどの Reducer へ渡すかを決めることができる。また、サンプリングをするだけでは、すべての Suffix の S-prefix となる語を得ることができないが、分割する境界となる語が得られることから、この語と辞書式順序でどちらが大きいかを計算することで、サンプリングから漏れている語を S-prefix に持つ Suffix についても適切に Reducer の割り当てを行うことができる。

5.4 各 Reducer 内での詳細 Suffix Array の構築

reduce 関数の入力については、第 4.3 節について述べたとおり、キーについては map 関数にて分割された Suffix、値については単語の出現位置のリストである。

第 4.3 節で述べたとおり、部分 Suffix Tree を生成する際、枝情報決定のために配列に一定数の単語を読み込む際に発生するファイル入出力を分散させることは難しい。そのため、この部分では分散はせず、reduce 関数の中で繰り返し、枝情報が決定できるまで配列に文字列を読み込み、Suffix をソートすることを行う。

この処理を行うことで reduce 関数の出力はソートされた位置情報の値リストとすれば、部分 Suffix Array を出力することができる。

6 評価実験

6.1 実験目的と方法

第 5 章にて説明した提案手法について、分散処理によってどの程度の時間が短縮できたか、またファイル入出力を削減したことによる時間に対する寄与を調べる実験を行った。

第 5 章で説明した提案手法で並列分散させる方式と、ベースライン手法について比較を行う。ベースライン手法は、ほぼそのまま実装したものである。すなわち、ERA アルゴリズムと同様に S-prefix のグループ化を行い、MapReduce の枠組みによる S-prefix のグループ分けを行わない。また、S-prefix の出現位置を枚挙を Reduce での集約を利用せずに、ERA アルゴリズムと同様に、元文書から探索を行うことで、枚挙する。つまり、map 関数では、第 4.2 節の手法の通り、文章内のすべての S-prefix について枚挙し、その S-prefix に対して主記憶容量に収まらない場合に単語を S-prefix に対して加えるようにし、どの Reducer にどの S-prefix を割り当てるかのグループ分けを行う。reduce 関数では、第 4.3 節の手法の通り、入力された S-prefix に対して、文章内での S-prefix の出現位置をすべて枚挙し、その位置からはじまる単語について読み取り、すべての S-prefix について辞書式順序になるようにソートすることを行う。表 4 に提案手法とベースライン手法の比較について示す。ベースライン手法と提案手法を比較すると、提案手法によりファイル入出力がどれくらい削減され、その結果として、処理時間がどれくらい短くなるのかがわかる。

表 4: ベースライン手法と提案手法の比較

	ベースライン	提案手法
S-prefix の集合の分割を partitioner に任せ、Map 分割を並列化	×	○
Reduce で S-prefix の出現位置列挙	○	×

実験に用いる入力文章は、横浜の地方議会の会議録コーパスより、40,000 行を切り出した文章とした。これは、約 10MB 程度のサイズである。40,000 行の入力文章内に含まれる単語の異なり数は 16,345 である。

Copyright © 2014 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.

表 1: 実行時間

<table>
<tr><th></th><th>ベースライン</th><th colspan="2">提案手法 (手法 A)</th><th>提案手法 (手法 B)</th></tr>
<tr><td>全体的な処理時間</td><td>211 分 39 秒</td><td colspan="2">14 分 31 秒</td><td>19 分 50 秒</td></tr>
<tr><td rowspan="3">Map 処理時間</td><td rowspan="3">146 分 0 秒</td><td>M0</td><td>24 秒</td><td rowspan="3">22 秒</td></tr>
<tr><td>M1</td><td>57 秒</td></tr>
<tr><td>M2</td><td>61 秒</td></tr>
<tr><td>Reduce 処理時間</td><td>65 分 18 秒</td><td colspan="2">13 分 45 秒</td><td>19 分 09 秒</td></tr>
</table>

表 2: ベースライン手法における各 Reducer での実行結果

ベースライン手法	R0	R1	R2	合計
入出力グループ数	1,287	4,889	10,169	16,345
入力レコード数	1,287	4,889	10,169	16,345
処理時間	18 分 56 秒	42 分 33 秒	65 分 18 秒	-

表 3: 提案手法における各 Reducer での実行結果

提案手法	R0	R1	R2	合計
入出力グループ数	1,286	4,303	10,756	16,345
入力レコード数	555,363	555,046	551,046	1,661,455
処理時間	12 分 29 秒	13 分 45 秒	13 分 01 秒	-

ベースライン手法では、Mapper 数を 1, Reducer 数を 3 とした。提案手法では、分散処理を行うことによってどれほど時間が変わってくるのかについて検証するために、Mapper 数を 3, Reducer 数を 3 とした手法 A と、Mapper 数を 1, Reducer 数を 1 とした手法 B の 2 つについて実験を行った。

評価尺度は分散が均等に行われたかどうかを考察するために Reducer の入力グループ数と処理レコード数を調べる。また、ファイル入出力や分散によってどの程度効率化できたかを調べるために、Map 処理、Reduce 処理、MapReduce 全体での実行時間を調べる。

6.2 実験結果

処理時間については表 1 に示す。各 Reducer での入出力グループ数、入力レコード数、処理時間は表 2、表 3 に示す。表 2 がベースライン手法に対応する。一方、表 3 は、Mapper 数が 3、Reducer 数が 3 である手法 A に対応する提案手法である。

表 1 より、提案手法のほうがベースライン手法に比べて実行時間が短くなっていることがわかる。つまり、ファイル入出力の回数を減らした提案手法のほうが、全体の速度が向上したと言える。また、手法 A と手法 B を比較すると、全体の処理時間は手法 A のほうが短いが、Map 処理は手法 A よりも手法 B のほうが処理時間が短いことがわかる。

6.3 考察

map 関数でのファイル入出力は、ベースライン手法では、単語集合を得る時、1 単語についての頻度を得る時の 2 箇所において発生する。そのため、map 関数では入出力レコード数と単語集合を得る 1 回のファイル入出力が少なくとも発生する。つまりこの実験においては 16,346 回のファイル入出力が削減できていることになる。また、reduce 関数においても、単語の出現位置について調べる際のファイル入出力が削減できていることになる。そのため、Reducer 全体で見ても 16,345 回のファイル入出力が削減できたことになる。その後の処理についてのファイル入出力数についてはベースライン手法、提案手法ともに同じ回数となる。各 Reducer について、入出力グループの数だけファイル入出力が発生することについて着目すると、表 2 と、表 3 の結果より、1 グループあたり 300 ミリ秒程度ファイル入出力に時間が発生していることが考えられる。このため、ファイル入出力を減らすことには大きな意義があることがわかった。提案手法において、他にファイル入出力が発生する箇所として、辞書順に並び替える際に、一定数文字列を読み込む箇所がある。この部分についてはランダムアクセスで読み取りに行くため、ファイルの先頭からのバイトオフセットが大きくてもあまり関係しない。

次にレコード数を均等にできているかについて確認すると、表 3 においてどの Reducer についてもレコード数は 550,000 件程度となっており、Partitioner については均等に各 Reducer へレコードを分けていることがわかる。しかし、実際の処理時間について見ると、入力レコード数については R0 よりも R1 のほうが若干程度だが遅いにもかかわらず、レコード数は R1 のほうが小さい。このため、入力グループの数も処理時間に関係があるのではないかと考えられる。MapReduce の Reduce への入力の際に、ある程度のオーバーヘッドが生じている

Copyright © 2014 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.

可能性も考えられる。

また、表 1 の結果より、提案手法の手法 A と手法 B の処理時間から、分散したことによる効果が少ないように見える。手法 A の Map 処理時間について見ると、Mapper によって処理時間が大幅に違っているように見える。本来、Mapper への入力に対して用いられている TextInputFormat での分割は、Hadoop 上のファイルシステムのブロックサイズであることから、今回の入力文章が 40,000 レコードという小さな単位で実行したことにもかかわらず、3 分割して分散処理を行ったため、ここでもある程度のオーバーヘッドが生じている可能性があると思われる。

7 結論

7.1 まとめ

本稿では、Hadoop を用いて Suffix Array を並列分散計算することで構築し、可変長 N-gram の頻度を求める手法を検討した。提案手法として、従来の並列計算可能な Suffix Tree 構築法を MapReduce のプログラミングモデルに沿って並列分散化を行い、さらにファイル入出力を削減することによって、効率的に Suffix Array を構築した。そしてこれを用いて可変長単語 N-gram の頻度について計算を行った。これを従来の方法のまま Suffix Array を構築した場合と比較し、評価実験を行った結果、提案手法はファイル入出力の削減および並列分散化により効率的に Suffix Array を構築できたと言える。

7.2 今後の課題

最後に今後の課題について述べる。

各 Reducer へのキー割当数の改善 本稿では、MapReduce プログラミングモデルに沿って、Suffx Array をまず構築した。Mapper での出力を Reducer へ割り当てる際に、各 Reducer での頻度数の合計が均等になるように実装した。この Reduce への入力の際に、ある程度のオーバーヘッドが生じている可能性も考えられる。このオーバーヘッドについて、原因を含め調査する必要があると考えられる。

Mapper 部分の連結する単語数の自動決定 本稿では、map 関数の処理において、ファイルアクセスを削減するために、入力文から分割する Suffix の単語数は 1 と仮定している。今後、入力文章について、Suffix の単語数が 2 以上になる場合を考えた時、連結する単語数は自動的に決定できることが望ましい。

Suffix Array 構築以外についての並列分散化 本稿での手法は主に Suffix Array の構築について並列分散化を行った。今後、入力文章に対して形態素解析を行う部分や、出力された Suffix Array を用いて可変長単語 N-gram の頻度を計算する部分についても入出力される文章が大規模であればあるほど時間がかかると考えられるため、これらの部分についても並列分散化することを検討すべきであろう。

参考文献

[1] 森信介, 土屋雅稔, 山地治, 長尾真. 確率的モデルによる仮名漢字変換. 情報処理学会研究報告. 自然言語処理研究会報告, Vol. 98, No. 48, pp. 93–99, (1998).

[2] 甲斐充彦, 廣瀬良文, 中川聖一. 単語 n-gram 言語モデルを用いた音声認識システムにおける未知語・冗長語の処理 (< 特集 > 音声言語情報処理). 情報処理学会論文誌, Vol. 40, No. 4, pp. 1383–1394, (1999).

[3] George Doddington. Automatic evaluation of machine translation quality using n-gram co-occurrence statistics. In *Proceedings of the Second International Conference on Human Language Technology Research*, HLT '02, pp. 138–145, San Francisco, CA, USA, (2002).

[4] 持橋大地, 隅田英一郎. 階層 pitman-yor 過程に基づく可変長 n-gram 言語モデル (知識処理,< 特集 > インタラクションの理解とデザイン). 情報処理学会論文誌, Vol. 48, No. 12, pp. 4023–4032, dec (2007).

[5] Graham Cormode and Marios Hadjieleftheriou. Methods for finding frequent items in data streams. *The VLDB Journal — The International Journal on Very Large Data Bases*, Vol. 19, No. 1, pp. 3–20, (2010).

[6] Mikio Yamamoto and Kenneth W. Church. Using suffix arrays to compute term frequency and document frequency for all substrings in a corpus. *Computational Linguistics*, Vol. 27, No. 1, pp. 1–30, (2001).

[7] 岡野原大輔. 高速文字列解析の世界ーデータ圧縮・全文検索・テキストマイニング. 岩波書店, (2012).

[8] The Apache Software Foundation http://hadoop.apache.org/. *Apache Hadoop*.

[9] Tom White. Hadoop 第三版. オライリー・ジャパン, (2013).

[10] Essam Mansour, Amin Allam, Spiros Skiadopoulos, and Panos Kalnis. Era: efficient serial and parallel suffix tree construction for very long strings. *Proceedings of the VLDB Endowment*, Vol. 5, No. 1, pp. 49–60, (2011).

[11] Makoto Nagao and Shinsuke Mori. A new method of n-gram statistics for large number of n and automatic extraction of words and phrases from large text data of japanese. *COLING '94 Proceedings of the 15th conference on Computational linguistics*, Vol. 1, pp. 611–615, (1994).

[12] Concurrent Inc. *Cascading, Application Platform for Enterprise Big Data http://www.cascading.org/.*

Copyright © 2014 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.