大規模コーパスを扱うためのツール群

岡野原 大輔 †　辻井 潤一 †‡§
† 東京大学情報理工学系研究科コンピュータ科学専攻
‡School of Computer Science, University of Manchester
§NaCTeM (National Center for Text Mining)
{ hillbig, tsujii }@is.s.u-tokyo.ac.jp

概 要

本稿では大規模コーパスを利用した研究を支援するためのツール群 **tx**, **bep**, **oll** を紹介する．近年，入手可能となってきた日本語 N グラムコーパスや Wikipedia などの大規模コーパスは，従来コーパスと比べ非常に大きく（数 GB〜数百 GB)，一般のマシン上で処理することが難しかった．そのため，従来の実装やライブラリを直接利用することができず，研究の妨げとなっていた．この問題を解決するため，我々は近年の圧縮全文索引技術やオンライン学習手法の研究成果を組み合わせ，こうした大規模コーパスを効率良く処理するためのツール群を開発した．本稿ではそれらのツールの実装詳細，および，それらを実際に利用した時の性能について述べる．

1　はじめに

近年，日本語 N グラムコーパス [1]，Wikipedia[2]，Medline などの大規模コーパスが利用可能となり，これらを基にした自然言語処理研究が盛んになってきている．

これらのデータは，新聞などを基にした従来のコーパスと比較し非常に大きい．例えば日本語 N グラムコーパスでは，ウェブ上に公開されている日本語データ（約 200 億文，2550 億単語）のうち，20 回以上出現した長さ 7 単語以下の全ての部分文字列の出現回数を記録しており，単語種類数のみでも約 250 万単語となる．

従来の自然言語処理は，比較的小規模のコーパスからどのようにして有益な情報を抽出し，高精度の分類器を構築するかに注力していたために，計算効率などは重要視されてこなかった．

これに対し，近年では前述のような大規模データが手に入るようになり，非常に大規模のデータからどのように有益な情報を効率よく抽出するかに注目が集まっている [3]．

機械学習の分野においても，大規模なデータを基にした学習を現実的な計算リソースで行なうための手法が盛んに研究されてきており，それらの成果は，オンライン学習，確率的勾配法，ランダムサンプリングなどでみることができる．我々は，これらの成果のうち計算量が特に少ないオンライン学習手法に注目し，それらをサポートしたライブラリ oll を開発した．

こうした学習手法と同様に問題となるのは，これらの情報をどのように保存し，処理するかである．これらを直接保存した場合は，高速な主記憶上に収まらないため，ディスクなど低速な二次記憶装置上で DB などを用いて保存したり，データをシーケンシャルに扱ったり，サンプリングなど近似的に処理することしかできない．

我々は，簡潔データ構造やアルゴリズムの最新の研究成果を活用することにより，大規模なデータを効率的に格納した上で，高速に処理可能なツール群，tx，bep を開発した．

これらツールは修正 BSD ライセンスに基づいて公開されている[1][2][3]．

本稿では，これら開発したライブラリの詳細を示すとともに，それらを実際のデータに適用した時の性能について述べる．

2　rank/select 辞書

初めに tx, bep で使われる簡潔データ構造 rank/select 辞書について説明する．

長さ n のビット配列 $B[0,\ldots,n-1]$ が与えられた時，rank/select 辞書は次の操作を備える．

- $\mathbf{rank}_b(B,i)$: $B[0,\ldots,i]$ 中の b の数を返す
- $\mathbf{select}_b(B,i)$: $(i+1)$ 番目の b の出現位置を返す

但し，$b \in \{0,1\}$ である．これらは $n + \mathrm{o}(n)$ ビットの作業領域を用いて定数時間で実現可能である（詳細は，[4] などを参照されたい）．

[1] tx: http://www-tsujii.is.s.u-tokyo.ac.jp/˜hillbig/tx-j.htm
[2] bep: http://www-tsujii.is.s.u-tokyo.ac.jp/˜hillbig/bep-j.htm
[3] oll: http://code.google.com/p/oll/wiki/OllMainJa

現在の tx，bep では rank/select 辞書の実現に，上記の時間，領域を達成するデータ構造を用いずに，より単純だが現実的には効率的な次の実装を利用する．

はじめに，配列 B を 256 個ずつの大ブロックに分割し，それぞれの大ブロックの先頭の $\mathbf{rank}$ の値を配列 $L[n/256]$ に格納する．次に各大ブロックを更に 32 個ずつの小ブロックに格納し，大ブロックの先頭から各小ブロックの先頭までに出現した 1 の数を配列 $S[n/32]$ に格納する．これらを用いて $\mathbf{rank}_1$ は，

$$\mathbf{rank}_1(B,i) = L[i/256]+S[i/32]+popcount(B,\lceil i/32\rceil,i)$$

と実現できる．ただし $popcount(B,i,j)$ は $B[i,\ldots,j]$ 中の 1 の数を返す関数であり，全ての 8 ビットの配列パターンについての答えを記録した配列を参照することにより定数時間で求めることができる[4]．$\mathbf{rank}_0(B,i) = i - \mathbf{rank}_1(B,i)$ であり，**select** は L, H を用いた二分探索を用いて行なう．L 中の各エントリはそれぞれ 4 バイトで保存し，S 中の各エントリは最大値が 256 であることから 1 バイトで格納可能である．よって，この rank/select 辞書は，ビット配列本体を含め合計で $1.375n$ ビットで格納可能である．

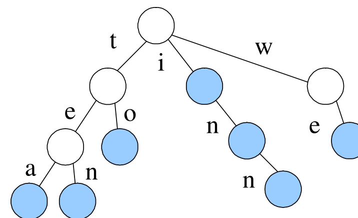
3 tx

tx は大規模なキー集合による辞書を管理し，その上で共通接頭辞探索など複雑なクエリを効率的に処理するためのライブラリである．内部実装に，trie データ構造の簡潔表現である LOUD（Level-Order Unary Degree Sequence）[5] を利用しており，キー集合自身をそのまま格納した場合の 1/4〜1/10 の作業領域量で，様々な操作を実現可能である．

はじめに trie データ構造について簡単に説明する．trie は節点と枝からなる木構造からなり，各枝には 1 文字が対応し，全ての葉と一部の節点は一つのキーに対応する．そして，根から葉に向かって枝を辿っていった時，枝に付いている文字をつなげたものが各キーの文字列表現に対応する．各キーに対応する値は，キーに従って辿り着いた最後の節点/葉に格納する．図 1 に trie の例を示す．

この木を利用することで，キー集合に対する次の各操作が効率的に実現できる．

- lookup : 与えれたキー k が辞書が含まれているかを調べる．また含まれているならば，そのキーに対応するデータを返す



キー : "to", "tea", "ten", "i", "in", "inn“, and “we”.

図 1: trie の例．色付きの節点/葉にキーに対応するデータを格納している．

- predictive search : 与えられたキー k を接頭辞として含むキーが辞書に含まれているかを調べる．また含まれているならば，該当する全てのキーを列挙し，それらに対応するデータを返す
- common prefix search : 与えられたキー k の接頭辞からなるキーが辞書に含まれているかを調べる．また含まれているならば，該当する全てのキー集合を列挙し，それらに対応するデータを返す

特に，2 番目，3 番目の操作についてはハッシュを用いた連想配列では効率的に処理できないことに注意する．

例えば，形態素解析においては，与えられた文に対し，その文に含まれる，辞書に登録されているキーを全て列挙する必要があるが，common prefix search を各位置で適用することで効率的に列挙できる．

この trie の実装には，木を直接ポインタで表現する方法が最も簡単だが，大規模キー集合を格納する場合，空間領域量が非常に大きくなる．例えば各ノードが，親ノード，次の兄弟ノード，最初の子ノードへのポインタを持っている場合，1 ノードあたり $3*32 = 96$ ビット必要となる[5]．

これに対し，tx が用いている LOUD 表現では 1 ノードあたり 2 ビットしか使わないながら，各節点から親，次の兄弟，子への遷移を全て定数時間で実現する．この作業領域量は，順序木を格納するのに必要な情報理論的な下限に漸近する．

LOUDS 表現は，木を，幅優先探索で各節点を辿りながら，d 個の子がある節点を，d 個の ‘1’ と 1 個の ‘0’ により表現したものをつなげることで表現する．これは各節点の子の数を単進表現している．また，このビット配列の先頭には，番人の役割をする Super Root: S(‘10’）を追加しておく．LOUDS 表現の例を図 2 に示す．

n 個の節点からなる木では，各節点では必ず ‘0’ が 1 回出力されるので，$n+1$ 個の 0 があり（Super Root を含めることに注意する），また各節点は，その親で 1 個の

[4]配列を用いる他にビット操作を組み合わせた方法や，直接 CPU 命令を利用する方法もある

[5]これはノード数が 2^{32} より小さい場合であり，これより大きい場合は，作業領域量はより大きくなる

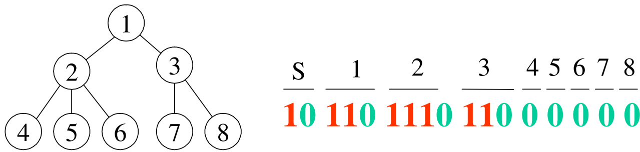


図 2: 木（左）の LOUDS 表現（右のビット配列）の例

'1' によって表現されているので n 個の 1 がある（Super Root は親が無いので含まれない）．よって，このビット配列の長さは $n+1+n=2n+1$ ビットである．このビット配列を $B[0,\ldots,2n]$ とおく．

幅優先探索で辿った時に k 番目に辿った節点は，B 上では k 番目の '1' に対応し，$k+1$ 番目の '0' に対応している．例えば図 2 において，左の木で 2 と番号が付いている節点は，'1' 表現では $B[2]$, '0' 表現では $B[8]$ が対応する．これら二つの表現は **rank**，**select** 操作を使って定数時間で変換可能であり，たとえば，'1' 表現で p の位置にある場合，それに対応する '0' の位置は $\mathbf{select}_0(\mathbf{rank}_1(B,p)-1)$ として求められる．

以降では '1' の位置で節点を表現することにする．i の位置にある "1" に対応する節点からの移動は **rank/select** 操作を利用し，次のように実現できる．

- 最初の子 = $\mathbf{select}_0(B,\mathbf{rank}_1(B,i)-1)+1$
- 次の兄弟 = $i+1$
- 親 = $\mathbf{select}_1(B,\mathbf{rank}_0(B,i)-1)$

そして，trie 表現中の枝に付属する文字を表現するために，木を表現する rank/select 辞書 B に加え，次のように定義される配列 $E[0,\ldots,n-1]$ とビット配列 $T[0,\ldots,n]$ を利用する．

配列 $E[i]$ には，$i+2$ 番目の節点に向かう時の枝に対応する文字を格納する．また $T[i]$ には，$i+1$ 番目のノードがキーの末端に対応するならば 1，そうでないならば 0 を格納する．

これら配列 B,T,E を利用することで，i 番目の節点に向かう時の枝に付属する文字を定数時間で参照することができる．E は $n\log_2\sigma$ ビット，T は n ビットで表現可能であり（但し σ は文字種類数），文字種類数が 256 の時，rank/select 辞書に必要な補助データ構造も含め約 $12n$ ビットで表現可能である．

3.1 tx を利用する際の工夫

tx の実装では，文字種類数が 256 を仮定しているが，日本語などの多バイト文字列の場合でも，UTF-8 など，文字が他の文字の途中で出現しないことが保障されていれば，そのまま tx を利用してキーワードを管理し，処理することが可能である．

Algorithm 1 Byte Aligned 符号

input $x \in N^+$
output $c[0\ldots size]$ x の符号結果
  $size = 0$
  while $x \geq 128$ **do**
    $c[size] = 128+$ (x の上位 7 ビット)
    $size = size + 1$
    x の上位 7 ビットを 0 にする
  end while
  $c[size] = x$
  $size = size + 1$

さらに，一般に文字の出現分布は非常に偏りが大きく，それを利用することで全体のサイズを小さくすることができる．ここでは，一番単純な Byte Aligned 符号を利用し作業領域量を小さくすることを考える．

はじめに，文字集合 Σ を出現頻度が大きい順にソートした文字集合 Σ' を考える．次に，Σ' の各文字 x の符号を図 1 のように決める．この符号は prefix-free であり，瞬時復号可能である．この符号を使って全てのキーを符号化する．この時，頻出する文字は短い符号長を持ち，全体として短い符号長が期待できる．　このように管理された trie 集合に対し，クエリを要求する場合は，入力クエリを同様に符号化し，得られた結果を復号すればよい．　この変換は日本語のみならず，英語の N グラムを管理する時などにも，各単語をソートし，符号化しておくことで効率的に扱うことができる．

次に，各キーに対応する値を格納したい場合だが，tx はキー集合 $k_1,\ldots,k_n$ の各キーに対し $[0,n-1]$ 中の一意な番号 (ID) を返すので，値を格納する長さ n の配列 V を用意し，得られた ID の場所に値を格納することで，管理することができる．これは後述する bep も同様にして値を格納することができる．

4 bep

bep も tx 同様に大規模なキー集合を扱うためのデータ構造である．bep は最小完全ハッシュ関数を利用したデータ構造であり，共通接頭辞探索など複雑な操作を備えていないが，キー自身を保存しなくて良い場合は，1 キーあたり約 4 ビットで保存することができ，キーをそのまま保存する場合に比べ，遥かに少ない作業領域量で保存することができる．

はじめに最小完全ハッシュ関数 (Minimal Perfect Hash Function: 以下 MPHF と呼ぶ) について説明する．まず，キー集合 $S=\{k_1,k_2,\ldots,k_n\}\subseteq U$ に対し $h: U\mapsto N$

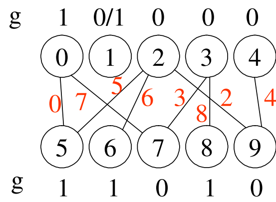


図 3: 最小完全ハッシュ関数の構成例．各キーが枝に対応し，枝の脇の数字が実際に割り当てられた頂点番号（ハッシュ値）．g については本文参照（0/1 はどちらでも良いという意味）

が $h(k_i) \neq h(k_j), 1 \leq i < j \leq n$ を満たしている時，これを完全ハッシュ関数（Perfect Hash Function: PHF）と呼ぶ．つまり，これから登録しようとしているキー同士で衝突が決して起こらないことが保障されているハッシュ関数のことである．

そして，S に対して h が MPHF であるとは，h が PHF であり，かつ，値域が $[0, m-1]$ である場合である．つまりハッシュ値が無駄なく，重複の無い連番を返す場合である．

この MPHF は，通常のハッシュ関数を直接用いて構成するのは難しく，m 個のキーで，最小完全ハッシュ関数が偶然得られる確率は $m!/m^m \approx e^{-m}$ である．

bep では，MPHF の構成手法に [6] で提案された方法を用いる．

はじめに，n 個のキー $S = \{k_1, k_2, \ldots, k_n\}$ に対し，$[0, m), n \leq m$ を値域とする PHF（MPHF では無いことに注意）を以下のように構成する．

まず，二つの独立なハッシュ関数 h_1, h_2 を用意し，これらの値域がそれぞれ $[0, m/2), [m/2, m)$ になるようにする．これらは一様でありさえすればよく，PHF である必要は無い．

次に，これらのハッシュ関数から定義される二部グラフ $G = (V_1, V_2, E)$ を考える．$V_1 = \{0, \ldots, m/2-1\}$, $V_2 = \{m/2, \ldots, m-1\}$ であり，各枝は各キー k_i に対応する $e_i = \{h_1(k_i), h_2(k_i)\}$ $(i = 1, \ldots, n)$（図 3 を参照）．

そして，各枝 e_i に対し，それが属する二つの頂点のどちらかを割り当てる．この時，複数の枝が同じ頂点を選ばないようにする．グラフ G に閉路が無い場合はこの割り当ては必ず求められる．グラフ G に閉路がある場合は，また新しいハッシュ関数を用いて，閉路が無いグラフ G ができるまで作りなおす．

枝 e_i が V_1 側に割り当てられた場合 $t(k_i) = 0$，V_2 側の場合 $t(k_i) = 1$ とした時，各頂点 v に対し，$g(v) \in \{0, 1\}$ を，$g(h_1(k_i)) + g(h_2(k_i)) \mod 2 = t(k_i)$ が全ての枝に成り立つように割り当てる．この割り当てはグラフ G に閉路が無いため必ず求められる．具体的には，次数が一番大きい頂点を探し，そこから枝を辿りながら各頂点の g を greedy に決定していくことにより求められる．

これにより全てのキーが，G 中の一つの頂点が対応することになり，この頂点番号をハッシュ値として考えることで PHF が完成する．キー k に対する最終的なハッシュ関数は次の通りである．

$$\begin{aligned} h(k) &= h_i(k) \\ i &= g(h_1(k)) + g(h_2(k)) \mod 2 \end{aligned} \quad (1)$$

構築時間で問題となるのは，グラフ G に閉路が無い場合である．$m = 2.09n$ の時，先程のように構成された G に閉路が無い確率は約 0.29 と知られており [6]，平均 3〜4 回作り直せば閉路が無いグラフ G が得られる．bep では，二つのハッシュ関数ではなく，三つのハッシュ関数を用いて構成されたハイパーグラフに対し，先程と同様に，各キーを一つの頂点に割り当てている．このハイパーグラフでは，$m = 1.23n$ の時，先程のように構成された G に閉路が無い確率はほぼ 1 である [6].

このように構築された PHF は既に値域中の大部分にキーが割り当てられており（1/1.23 がキー），MPHF はキーが割り当てられている場所を列挙し番号を順に与えるだけでよく，これには前述の rank 関数を利用する．配列 $B[0, m-1]$ を，i にキーが割り当てられている場合 $B[i] = 1$，それ以外 $B[i] = 0$ と定義すると $\mathbf{rank}_1(B, i) - 1$ により定義される関数は $[0, n)$ を返す．

最終的な MPHF の形は次の通りである．

$$\begin{aligned} h(k) &= \mathbf{rank}_1(B, h_i(k)) - 1 \\ i &= g(h_1(k)) + g(h_2(k)) + g(h_3(k)) \mod 3 \end{aligned} \quad (2)$$

保存に必要な作業領域量は，$g[0, m)$ の保存に $2m$ ビット，$B[0, m)$ の保存に m ビットであり，合計で $3m = 3.69n$ ビット必要である．

MPHF の特性として，S に含まれていないキーが与えられても，必ず $[0, n)$ 中の値を返してしまうことが挙げられる．つまり，キーが含まれないことを判定することは MPHF だけではできない．そのため，キーの存在判定を行なうためには MPHF に加え，登録したキー自身を保存する必要がある．この場合，作業領域量はキーそのまま保存する分だけ増えることになる．

今後の課題として，Bloomfilter などと組み合わせることにより，辞書に含まれないキーに対しては非常に高い確率で含まれないと報告できるようなデータ構造を作る予定である．

5 oll

oll は様々なオンライン学習手法をサポートしたライブラリであり，現在，Perceptron [7](以下 P)，Averaged Perceptron [8] (AP), Passive-Agressive [9] (PA, PA-I, PA-II), Confidence-Weighted [10] (CW) をサポートしている．

全てのデータに対し最適化を行なうバッチ学習に対し，オンライン学習ではデータの部分集合から統計値を取り出し，それを利用し最適化を行なう．訓練データに冗長性があり，全てのデータを見る前に最適化を行なうことができる場合は，オンライン学習を利用することにより効率的に学習を行なうことができる．また訓練データを全て保存しておく必要は無く，ストリーミングデータのようにデータを一度しか見ないですぐ捨てることも可能である．

oll では，様々な手法が同一形式で学習できるようになっており，学習手法間の比較が簡単に行なえるようになっている．

これら全ての手法は，線形識別器であり，モデルパラメータである重みベクトル $\boldsymbol{w} \in R^m$ を利用して，入力ベクトル $\phi(x)$ に対し，$\boldsymbol{w}^T\boldsymbol{\phi(x)}$ の符号を調べることで分類を行なう点で同じである．

これらの手法は，学習時の重みベクトルの更新式のみが違うだけである[6]．表 1 に各手法の更新条件，更新式，分類式を挙げる．

oll は訓練データが疎（多くの次元の値が 0）の場合に最適化されており，発火している素性数に比例する時間で更新，分類を行なうことができる．

今後は確率的勾配降下法や L_1-ball projection [11], larank[12]，Logistic 回帰のオンライン学習 [13] もサポートする予定である．

6 実験

本章では，これらのツールを実際のデータに適用した時の性能について述べる．

初めにキーを格納する手法である tx, bep についてその他の格納手法と比較を行なった．

実験データとして，Web 1T 5-gram Version 1 の 1-gram を利用した．キーワード種類数は 13588391 であり，キーワードを全てつなげた時の総長は，112349445 バイトであった．これらにより構築した $trie$ 中の葉も含めたノード数は 34722820 であった．

比較対象として，double array により trie を実装した darts [14]，C++ stl の set（赤黒木による実装）, hash_set を用いた．

但し，tx, darts 以外は lookup 操作しかサポートしていないことに注意する．

実行環境は CPU は 3GHz Xeon，主記憶は 32GB，g++ のバージョンは 4.03 で-O3 オプションを用いた．メモリ使用量については memusage コマンドを利用して heap peak+stack peak を計測した．

tx は，全てのキーを保存しながら作業領域量はキー自体を保存した場合の約半分を実現しており（キー集合自身の圧縮としても利用できる），従来の効率的な Trie の実装（例えば Double Array）などと比べて 1/10 の作業領域量を実現している．その代わり，一つの枝を辿る際に darts などと比べ複数の操作が必要であり，計算時間は darts の約 3 倍から 10 倍かかった．

bep は，参照時間は非常に速く，また作業領域量もキーを保存しない場合は非常に小さく保存できることが分かった．構築時間に関してはどの手法もほぼ同様であった．

次にオンライン学習ツールの oll を利用した文書分類タスクを行い，性能評価を行なった．データは libsvm の binary dataset 中の news20.binary[7]をそのまま用いた．クラス数は 2，データ数は 19996，素性種類数は 1355191 であった．これらのデータをシャッフルし，15000 例の訓練データと 4996 例のテストデータに分けた．バッチ学習の例として tiny SVM(以下 SVM)[8]を用いた．なお，SVM や最大エントロピー法などでも近年バッチ学習の高速化がされており，これらとの比較は今後の課題である．

PA-I，PA-II，CW，SVM におけるハイパーパラメータ C のチューニングは特に行なっていない（$C = 1.0$ を用いた)

初めにデータ繰り返し数を 10 回に制限した時の結果を表 3 に示す．ほぼ全ての手法が同様の精度を達成していることが分かる．特に，P， AP を除いた手法では SVM も含め，ほぼ差が殆どみられなかった．学習時間は，CW, SVM 以外はほぼ全て 0.5 秒と非常に高速であった．これは，1 訓練データを一回操作するのに 3 マイクロ秒しかかかっておらず数百万規模の訓練データにも十分対応できることが分かった．

次に，訓練データを保存できない場合などのケースで，訓練データを一度しか見ないで学習した場合の比較を表 4 に示す．殆どの学習器がほぼ先程の結果に近い結果を出しており，特に CW の性能は 1 回の繰り返しでほぼ収束していることが見られた．

[6] bias 項については，入力ベクトル $\phi(x)$ を $[\phi(x)^T, b]^T$ と拡張することにより考慮している

[7] http://www.csie.ntu.edu.tw/ cjlin/libsvmtools/datasets/binary.html#news20.binary

[8] http://chasen.org/ taku/software/TinySVM/

表 1: 各学習手法による各訓練例 $((\phi(x), y))$ が与えられた時の更新条件，更新式，分類式．$A+ = B$ は $A := A + B$ の略記である．$s = y\boldsymbol{w}^T\phi(x), t$ はそれまでに見た訓練例の数，γ は [10] 参照.

手法	更新条件	更新式	分類式
Perceptron [7]	$s < 0$	$\boldsymbol{w}+ = y\phi(x)$	$\boldsymbol{w}^T\phi(x)$
Averaged Perceptron [8]	$s < 0$	$\boldsymbol{w}+ = y\phi(x) \quad \boldsymbol{w}_a+ = y\frac{\phi(x)}{t}$	$(\boldsymbol{w} - \frac{\boldsymbol{w}_a}{t})^T\phi(x)$
PA [9]	$s < 1$	$\boldsymbol{w}+ = y\frac{1-s}{\lvert\phi(x)\rvert}\phi(x)$	$\boldsymbol{w}^T\phi(x)$
PA-I [9]	$s < 1$	$\boldsymbol{w}+ = y\min(C, \frac{1-s}{\lvert\phi(x)\rvert}\phi(x))$	$\boldsymbol{w}^T\phi(x)$
PA-II [9]	$s < 1$	$\boldsymbol{w}+ = y\frac{1-s}{\lvert\phi(x)\rvert+2C}\phi(x)$	$\boldsymbol{w}^T\phi(x)$
CW [10]	$\gamma > 0$	$\boldsymbol{w}+ = y\gamma\Sigma\phi(x) \quad \Sigma^{-1}+ = 2\gamma C\boldsymbol{diag}(\phi(x))$	$\boldsymbol{w}^T\phi(x)$

表 2: 各格納手法による性能比較．列 “complex search ” は，共通接頭辞検索などを備えているかを示す.

実装手法	サイズ（Byte)	入力サイズ比	構築時間 (秒)	lookup（10^6 回, 秒）	complex search
tx	52626805	0.46	30.5	10.3	○
bep	221056589	2.03	44.5	0.49	×
bep (キーチェック無し)	7452396	0.07	44.5	0.46	×
darts 0.31	406358432	3.62	16.7	1.25	○
stl::set	995599278	8.86	39.5	3.30	×
stl::hash_set	919761665	8.18	32.4	0.54	×

表 3: 各手法の結果．繰り返し回数が 10 回の時（SVM を除く）.

手法	P	AP	PA	PA1	PA2	CW	SVM (linear)
学習時間 (秒)	0.54	0.56	0.58	0.59	0.60	1.39	1122.60
精度 (%)	94.696	95.336	96.477	96.457	96.457	96.497	96.237

表 4: 各手法の学習結果．繰り返し回数が 1 回の時

手法	P	AP	PA	PA1	PA2	CW
学習時間 (秒)	0.05	0.09	0.07	0.08	0.08	0.21
精度 (%)	93.355	94.035	96.157	96.137	95.917	96.437

7 まとめ

本稿では，大規模な自然言語情報を処理するためのツールを紹介した．これらは基本的には既存研究の成果を組み合わせて作成されているが，いくつかは単純ではない実装上の工夫を行い，実用レベルにしている．

これらのツールは，修正 BSD ライセンスに基づいて公開されており，既にいくつかの研究グループや会社などで使われていて，実用性も評価されはじめている．

今後はこれらのツールを他の言語からも使いやすくすることや，機能追加，性能向上などが考えられる．特に構築時に必要な作業領域量を減らす工夫などは既存のツールのみから行なうのは困難であり、これらをサポートする必要がある．

また，これらのツールを様々な自然言語処理のタスクで利用し，これまでのスケールでは難しかった問題を解いていきたいと考えている．

参考文献

[1] 工藤拓 and 賀沢秀人. Web 日本語 n グラム第 1 版. 言語資源協会発行.

[2] http://ja.wikipedia.org/.

[3] Learning using many examples. NIPS 2007 Tutorial. http://leon.bottou.org/talks/largescale.

[4] P. Ferragina and G. manzini. Indexing compressed texts. *Journal of the ACM*, 52(4):552–581, 2005.

[5] O. Delpratt, N. Rahman, and R. Raman. Engineering the louds succinct tree representation. In *WEA*, pages 134–145, 2006.

[6] N. Ziviani F.C Botelho, R. Pagh. Simple and space-efficient minimal perfect hash functions. In *WADS*, 2007.

[7] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Cornell Aeronautical Laboratory, Psychological Review*, 65(6):386–408, 1958.

[8] M. Collins. Discriminative training methods for hidden markov models: Theory and experiments with perceptron algorithms. In *Proc. of EMNLP*, 2002.

[9] K. Crammer, O. Dekel., J. Keshet, S. Shalev-Shwartz, and Y. Singer. Online passive-aggressive algorithms. *JMLR*, 2006.

[10] M. Dredze, K. Crammer, and F. Pereira. Confidence-weighted linear classification. In *Proc. of ICML*, 2008.

[11] J. Duchi, S. Shalev-Shwartz, Y. Singer, and T. Chandra. Efficient projections onto the l1-ball for learning in high dimensions. In *Proc. of ICML*, 2008.

[12] A. Bordes, L. Bottou, P. Gallinari, and J Weston. Solving multiclass support vector machines with larank. In *Proc. of ICML*, pages 89–96, 2007.

[13] S. Shalev-Shwartz and Y. Singer. Tutorial on theory and applications of online learning. Tutorial ICML, 2008.

[14] http://www.chasen.org/ taku/software/darts/.