Implementing Deplump

Nicholas Bartlett Frank Wood

Department of Statistics, Columbia University, New York, USA

Abstract

1 Introduction

Deplump [Gasthaus et al., 2010] is a general purpose, streaming lossless compressor based on a probabilistic model of discrete sequences called the sequence memoizer [Wood et al., 2009]. Gasthaus et al. showed that although deplump is algorithmically similar to the PPM and CTW compression algorithms, particularly their unbounded context lengths variants [Cleary and Teahan, 1997; Willems, 1998], the coherent probabilistic model underlying deplump gives it a consistent empirical advantage. In particular, Gasthaus et al. showed that deplump generally outperformed CTW [Willems, 2009], PPMZ [Peltola and Tarhio, 2002], and PPM* [Cleary and Teahan, 1997] on the large Calgary corpus, the Canterbury corpus, Wikipedia, and Chinese text. Deplump was shown to underperform in comparison to the PAQ family of compressors [Mahoney, 2005], but the point was made that deplump (or more specifically the sequence memoizer) could replace one or all of the mixed, finite-order Markov-model predictors included in PAQ.

When introduced in [Gasthaus et al., 2010], deplump was reposed on a sequence memoizer whose space complexity was linear in the length of the input stream, rendering deplump inappropriate for stream compression. Since then, constant space approximations to the sequence memoizer have emerged. The sequence memoizer is an incremental method for hierarchically smoothing conditional distribution estimates when those conditional distributions are related to each other in a tree-shaped graphical model. The space complexity of the sequence memoizer is a function of the number of instantiated nodes in the corresponding graphical model and the storage required at each node. Bartlett et al. introduced an approximation to the sequence memoizer in which the number of nodes in the tree is of asymptotically constant order [Bartlett et al., 2010]. Unfortunately, in their work the storage requirement at each node grew as an uncharacterized but not-constant function of the input sequence length. **?** independently introduced a method for constraining the memory used at each node in the tree to be of constant order [**?**] but did so in a way that requires the computational cost of inference to grow as a super-linear function of the length of the training sequence.

The primary contribution of this work is to marry these two separate bodies of work such that constant memory, linear time approximate inference in the sequence memoizer is achieved, thereby rendering deplump appropriate for streaming lossless compression

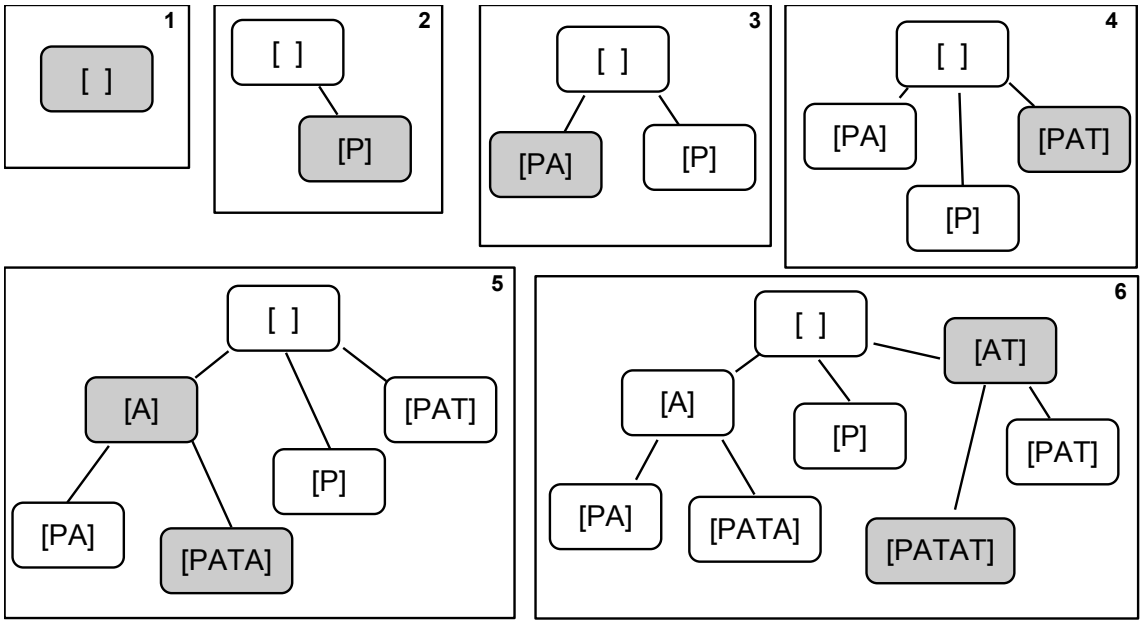

Figure 1: Construction of suffix tree for string "PATAT". In each frame the new nodes are shaded in gray.

applications without giving up the its inherent advantages while retaining. Additionally, we have aimed to make a comprehensive algorithmic description of deplump available to those who might choose to implement deplump for their own purposes. This accompanies a reference implementation which is available on the `http://www.deplump.com/` website.

2 Previous Work

A lossless compressor based on a probabilistic model called the Sequence Memoizer (Wood et al. [2009]) was proposed in Gasthaus et al. [2010]. Empirical performance was evaluated on the calgary corpus and an algorithm was outlined. Unfortunately, In Gasthaus et al. [2010] the algorithm proposed has a memory footprint which grows linearly in the length of the input sequence, making it unsuitable for the compression of streams. A constant space algorithm for inference in the Sequence Memoizer model has since been developed by Bartlett et al. [2010]. This implementation of Deplump builds on the inference algorithms proposed in Bartlett et al. [2010] and Gasthaus et al. [2010] to develop a compressor which has finite memory complexity.

3 Algorithm

Given an ordered symbol set Σ, probabilistic compression algorithms work by using a probabilistic model to predict a sequence of symbols. The predictive distribution function

is then used as the parameter in a range encoder to compress the stream. The details of a range encoder implementation are not included here, we only note that if the predictive distribution function is F and the next symbol in the stream is s then a range encoder takes $F(s-1)$ and $F(s)$ as arguments and returns a bit stream (possibly null). To decompress, the range decoder takes F and the compressed stream as arguments and returns the next symbol in the uncompressed sequence. In the algorithm the functions RangeEncode() and RangeDecode() indicate these operations. The use of a cumulative distribution function is well defined since the symbols are ordered. The notation $s-1$ refers to the symbol prior to s in the symbol ordering. In order to decompress the stream the exact same predictive model will need to be built from the compressed stream. This requires that the model estimate prior to compressing s_n is a function of fixed parameters and the symbols $[s_0, s_1, \ldots, s_{n-1}]$ because those are the only symbols available to the decompressor for decompressing s_n.

The algorithm operates on a suffix tree. A suffix tree is a data structure for keeping track of the unique suffices in a set of strings. The tree structure arranges the suffices hierarchically which makes it easy to search. In the case of a single stream the set of strings to consider is the set $\{[], [s_0], [s_0, s_1], [s_0, s_1, s_2], \ldots\}$, which we refer to as the set of contexts. Each node of the suffix tree corresponds to a string of the form $[s_m, \ldots, s_{m+k}]$. In general we use $\mathcal{N}$ to refer interchangeably to a node instance and the context to which the node corresponds. The function CreateNode($\mathcal{N}, \mathcal{M}$) makes the creation of node $\mathcal{N}$ with parent $\mathcal{M}$ explicit. The parent of node $\mathcal{N}$ is referenced as PA($\mathcal{N}$).

Each node instance $\mathcal{N}$ contains two counts for each $s \in \Sigma$, c_s and t_s. We use c and t to refer to the marginal counts $\sum_{s\in\Sigma} c_s$ and $\sum_{s\in\Sigma} t_s$. Each node also has a discount d associated with it. The discount associated with $\mathcal{N}$ is a function of $\mathcal{D}$ (the discount parameters of the model), $|\mathcal{N}|$, and $|\text{PA}(\mathcal{N})|$. The discount for $\mathcal{N}$ is calculated by GetDiscount($\mathcal{N}$). Suffix tree data structures use a reference sequence ($\mathcal{RS}$) to store the unique suffices in the tree. Therefore, each node instance also contains two indices related to $\mathcal{RS}$ from which the context specific to that node can be reconstructed. If the indices for $\mathcal{N}$ are i and j, then the context associated with $\mathcal{N}$ is $\mathcal{RS}[i:j]$.

The reference sequence grows with the length of the input sequence and must be shortened as the algorithm progresses. The shortening of $\mathcal{RS}$ is made explicit by the σ function in CDFNextSymbol. The function $\sigma(\mathcal{S})$ returns $\mathcal{S}[2:\text{end}]$. When $\mathcal{RS}$ is shortened, nodes in the suffix tree which reference removed sections are no longer usable and must be removed from the tree to prevent a memory leak. To facilitate the removal process pointers are maintained from the elements of $\mathcal{RS}$ to the suffix tree nodes which reference them. Without the use of pointers, deletion of the unusable nodes requires a search over the tree which is prohibitive for large trees. The cost of these operations can be amortized by shorting $\mathcal{RS}$ in chunks and keeping pointers from each chunk of $\mathcal{RS}$ instead of each element. To minimize the impact of rendering nodes unusable by shortening the reference sequence, suffix tree nodes are updated as the algorithm progresses to reference recent sections of $\mathcal{RS}$.

Several parameters required by the model must be assigned at initialization. They are $\mathcal{D}$, k, L, η, and depth. $\mathcal{D} = [\delta_0, \delta_1, \ldots, \delta_{10}, \alpha]$ is a list of discount parameters, each taking a real value in $(0, 1)$. The parameter k is positive integer valued and is an upper bound on the total count c in each node. The parameter L is an upper bound on the number of node instances in the suffix tree. We use $100L$ as an upper bound on the length of $\mathcal{RS}$, but there is no explicit reason to tie these two upper bounds together. The parameter η is a learning

rate for the updating of the discount parameters and is typically set very small. Finally, the depth of the tree can be limited by a positive integer. Depth is not considered in Algorithm 1, but it could be incorporated into the function GetNode.

Since the model must be estimated incrementally, the suffix tree must also be incrementally constructed. Construction of the tree is handled by the function GetNode and FragmentNode in Algorithm 1. An illustration of the incremental construction can be seen in Figure 1 for the toy sequence [PATAT]. In frame 4 the function GetNode assigns [ ] to $\mathcal{M}$ and then [PAT] to $\mathcal{S}$ with $\mathcal{M} = \text{PA}(\mathcal{S})$. In Frame 5 GetNode assigns [PA] to $\mathcal{M}$, but then must assign [A] = FragmentNode($\mathcal{M}$) to $\mathcal{P}$ and PA($\mathcal{M}$) to $\mathcal{P}$. Node $\mathcal{S}$ is then created by CreateNode([PATA], $\mathcal{P}$). In each frame the first step is to find $\mathcal{M}$, which can be achieved by descending an appropriate path of the suffix tree. All of the nodes on the path to $\mathcal{M}$ and possibly $\mathcal{M}$ itself can have the indices into $\mathcal{RS}$ updated to point to a more recent section of the reference sequence.

For each s in the input sequence the function PMFNextSymbol is called to obtain the predictive probability mass function (PMF). The function PMFNextSymbol enforces the upper bounds on $|\mathcal{RS}|$ and nc. If $|\mathcal{RS}|$ is hitting the upper bound then then $\mathcal{RS}$ is shortened as described, if nc is hitting the upper bound then leaf nodes of the suffix tree are deleted uniformly at random. Note that the deleting of leaf nodes uniformly at random is non-trivial to implement. One approach is to maintain an active list must of all the leaf nodes throughout the algorithm. A second approach is for each node to maintain a count of thenumber of leaf nodes in the subtree beneath it. Given that each node contains the exact number of leaf nodes beneath it a random leaf node can be obtained by taking a weighted random path down the tree. Finally, PMFNexSymbol returns the predictive PMF.

After encoding or decoding using the predictive PMF pi the first step in updating the model estimate is performed by the function UpdateCountsAndDiscounts. Starting at node $\mathcal{N}$ and progressing up to the root of the tree, c_s is incremented if t_s was incremented in the node below. If c_s is incremented, a stochastic decision is made to increment t_s. The gradients for the discount parameters $\mathcal{D}$ are updated by the function UpdateDiscountParameterGradients. If c is larger than k in any of the nodes, the counts c_s and potentially t_s are reduced by the function ThinCounts. Finally, $\mathcal{D}$ is updated based on the calculated gradients and the specified learning rate η and the symbol is appended to $\mathcal{RS}$.

4 Experiments

The empirical performance of Deplump was evaluated using the most recent Wikipedia content dump as a test corpus.

to evaluate the effect of various parameter settings on the compression performance of Deplump. The empirical per

Algorithm 1 Deplump

procedure DEPLUMP/PLUMP($\mathcal{IS}$)
  $\mathcal{RS} \leftarrow [\,]$ ▷ reference sequence
  Initialize $[\,]$ node of $\mathcal{T}$ ▷ suffix tree
  $nc \leftarrow 1$ ▷ node count
  $\mathcal{D} \leftarrow \{\delta_0, \delta_1, \delta_2, \ldots, \delta_{10}, \alpha\}$ ▷ discount parameters
  $\mathcal{G} \leftarrow \vec{0}$ ▷ discount parameter gradients, $|\mathcal{G}| = |\mathcal{D}|$
  $\mathcal{OS} \leftarrow [\,]$ ▷ output sequence
  for i = 1: $|\mathcal{IS}|$ **do**
    $[\pi, \mathcal{N}] \leftarrow$ PMFNextSymbol($\mathcal{RS}$)
    if Plump **then**
      $s \leftarrow$ RangeDecode(π, $\mathcal{IS}$)
      $\mathcal{OS} \leftarrow [\mathcal{OS}\ s]$
    else
      $s \leftarrow \mathcal{IS}[i]$
      $b \leftarrow$ RangeEncode($\Sigma_{i=1}^{s-1}\pi_i$, $\Sigma_{i=1}^{s}\pi_i$)
      $\mathcal{OS} \leftarrow [\mathcal{OS}\ b]$
    end if
    UpdateCountsAndDiscountGradients($\mathcal{N}$, s, π_s,TRUE)
    $\mathcal{D} \leftarrow \mathcal{D} + \mathcal{G}\eta/(\pi_s)$ ▷ update discount parameters
    $\mathcal{G} \leftarrow \vec{0}$ ▷ reset gradients to zero
    $\mathcal{RS} \leftarrow [\mathcal{RS}\ s]$ ▷ append symbol to reference sequence
  end for
  return $\mathcal{OS}$
end procedure
function PMFNEXTSYMBOL($\mathcal{RS}$)
  while $|\mathcal{RS}| \geq 100L$ **do**
    Delete nodes referencing $\mathcal{RS}$[1] and update nc
    $\mathcal{RS} \leftarrow \sigma(\mathcal{RS})$
  end while
  while $nc > (L-2)$ **do**
    Delete leaf node uniformly at random
    $nc \leftarrow nc - 1$
  end while
  $\mathcal{N} \leftarrow$ GetNode($\mathcal{RS}, \mathcal{T}$)
  $\pi \leftarrow$ PMF($\mathcal{N}, \vec{0}, 1.0$) ▷ $|\vec{0}| = |\Sigma|$
  return $[\pi, \mathcal{N}]$
end function

Algorithm 2 Deplump Continued

function GETDISCOUNT($\mathcal{N}$)
  $d = 1.0$
  if $\mathcal{N}$= [ ] **then**
    return δ_0
  end if
  for $i = (|\text{PA}(\mathcal{N})| + 1) : |\mathcal{N}|$ **do**
    if $i \leq 10$ **then**
      $d \leftarrow d\delta_i$ ▷ multiply by discount parameter i
    else
      $d \leftarrow d\delta_{10}^{\alpha^i}$
    end if
  end for
  return d
end function
function GETNODE($\mathcal{S}$, $\mathcal{T}$)
  Find the node $\mathcal{M}$ in the suffix tree sharing the longest suffix with $\mathcal{S}$.
  if $\mathcal{M}$ is a suffix of $\mathcal{S}$ **then**
    if $\mathcal{S} = \mathcal{M}$ **then**
      return $\mathcal{M}$
    else
      $\mathcal{S} \leftarrow$ CreateNode$(\mathcal{S}, \mathcal{M})$
      $nc \leftarrow nc + 1$
      return $\mathcal{S}$
    end if
  else
    $\mathcal{P} \leftarrow$ FragmentNode($\mathcal{M}$, $\mathcal{S}$)
    $\mathcal{S} \leftarrow$ CreateNode$(\mathcal{S}, \mathcal{P})$
    $nc \leftarrow nc + 1$
    return $\mathcal{S}$
  end if
end function

Algorithm 3 Deplump Continued

function UPDATECOUNTSANDDISCOUNTS($\mathcal{N}, s, p$, BackOff)
  $d \leftarrow$ GetDiscount($\mathcal{N}$)
  $pp \leftarrow p$
  if $c > 0$ **then**
    $pp \leftarrow (p - \frac{c_s - t_s d}{c}))(\frac{c}{td})$
    $w \leftarrow c_s + d(t * pp - t_s)$
  end if
  if BackOff and $c > 0$ **then**
    $c_s \leftarrow c_s + 1$
    BackOff $\leftarrow 0$
    BackOff $\leftarrow 1$ w.p. $pp(\frac{td}{w})$ ▷ w.p abbreviates "with probability"
    if BackOff **then**
      $t_s \leftarrow t_s + 1$
    end if
  else if BackOff **then**
    $c_s \leftarrow c_s + 1$
    $t_s \leftarrow t_s + 1$
  end if
  UpdateDiscountParameterGradients(t_s, t, pp, d)
  UpdateCountsAndDiscounts(PA($\mathcal{N}$), s, pp, BackOff)
  ThinCounts($\mathcal{N}$)
end function
function THINCOUNTS($\mathcal{N}$)
  $d \leftarrow$ GetDiscount($\mathcal{N}$)
  while $c > k$ **do**
    $s \leftarrow$ DrawMultinomial(π) s.t. $\pi_l = \frac{c_l}{c}$ ▷ π is a distribution over Σ
    $\phi \leftarrow$ SamplePartition(c_s, t_s, d)
    $i \leftarrow$ DrawMultinomial($[\frac{\phi_1}{c_s}, \frac{\phi_2}{c_s}, \ldots, \frac{\phi_{t_s}}{c_s}]$)
    if $\phi_i == 1$ **then**
      $t_s \leftarrow t_s - 1$
    end if
    $c_s \leftarrow c_s - 1$
  end while
end function

Algorithm 4 Deplump Continued

function PMF($\mathcal{N}, \pi, m$)
  $d \leftarrow$ GetDiscount($\mathcal{N}$)
  if $c > 0$ **then**
    for $s \in \Sigma$ **do**
      $\pi_s \leftarrow \pi_s + m(\frac{c_s - t_s d}{c})$
    end for
  end if
  if PA($\mathcal{N}$) $\neq$ null **then**
    return PMF(PA($\mathcal{N}$), π, dm)
  else
    $\pi \leftarrow (1 - dm)\pi + dm\mathcal{U}(\Sigma)$ ▷ $\mathcal{U}(\Sigma)$ is the uniform distribution over Σ
    return π
  end if
end function
function FRAGMENTNODE($\mathcal{M}, \mathcal{S}$)
  $d^{\mathcal{M}} \leftarrow$ GetDiscount($\mathcal{M}$)
  $\mathcal{P} \leftarrow$ maximum overlapping suffix of $\mathcal{M}$ and $\mathcal{S}$
  $\mathcal{P} \leftarrow$ CreateNode($\mathcal{P}$, PA($\mathcal{M}$))
  $nc \leftarrow nc + 1$
  PA($\mathcal{M}$) $\leftarrow \mathcal{P}$
  $d^{\mathcal{P}} \leftarrow$ GetDiscount($\mathcal{P}$)
  for $s \in \Sigma$ **do**
    $\phi \leftarrow$ SamplePartition $(c_s^{\mathcal{M}}, t_s^{\mathcal{M}}, d^{\mathcal{M}})$
    $t_s^{\mathcal{P}} \leftarrow t_s^{\mathcal{M}}$
    $t_s^{\mathcal{M}} \leftarrow 0$
    for $i = 1 : |\phi|$ **do**
      $a \leftarrow$ DrawCRP($\phi[i], d^{\mathcal{M}}/d^{\mathcal{P}}, -d^{\mathcal{M}}$)
      $t_s^{\mathcal{M}} \leftarrow t_s^{\mathcal{M}} + a$
    end for
    $c_s^{\mathcal{P}} \leftarrow t_s^{\mathcal{M}}$
  end for
  return $\mathcal{P}$
end function
function DRAWCRP(n, d, c) ▷ $n \geq 1$
  $t \leftarrow 1$
  for i = 2 : n **do**
    $r \leftarrow 0$
    $r \leftarrow 1$ w.p. $\frac{td+c}{i-1+c}$
    $t \leftarrow t + r$
  end for
  return t
end function

Algorithm 5 Deplump Continued

function SAMPLEPARTITION(c, t, d)
  $M \leftarrow t \times c$ matrix of zeros
  $M(t, c) \leftarrow 1.0$
  for $j = (c-1) : 1$ **do**
    for $i = 1 : (t-1)$ **do**
      $M(i, j) \leftarrow M(i+1, j+1) + M(i, j+1)(j - id)$
    end for
    $M(d, j) \leftarrow M(t, j+1)$
  end for
  $\phi \leftarrow \vec{0}$ ▷ $|\vec{0}| = t$
  $\phi[1] \leftarrow 1$
  $k \leftarrow 1$
  for j = 2 : c **do**
    $M(k, j) \leftarrow M(k, j)(j - 1 - kd)$
    $r \leftarrow 0$
    $r \leftarrow 1$ w.p. $\frac{M(k+1,j)}{M(k+1,j)+M(k,j)}$
    if r = 1 **then**
      $k \leftarrow k + 1$
      $\phi[k] \leftarrow 1$
    else
      $i \leftarrow$ DrawMultinomial($[\frac{\phi[1]-d}{j-1-kd}, \frac{\phi[2]-d}{j-1-kd}, \ldots, \frac{\phi[k]-d}{j-1-kd}]$)
      $\phi[i] \leftarrow \phi[i] + 1$
    end if
  end for
  return ϕ
end function
function UPDATEDISCOUNTPARAMETERGRADIENTS($\mathcal{N}, t_s, c, t, pp, d, m$)
  if $c > 0$ **then**
    if $|\mathcal{N}| = 0$ **then**
      $\psi \leftarrow \frac{1.0}{\delta_0}$
      $\mathcal{G}_0 \leftarrow \mathcal{G}_0 + (d(t * pp - t_s)\psi/c)m$
    else
      $z \leftarrow |\text{PA}(\mathcal{N})| + 1$
      while $z \leq |\mathcal{N}|$ and $z < 10$ **do**
        $\psi \leftarrow \frac{1.0}{\delta_z}$
        $\mathcal{G}_z \leftarrow \mathcal{G}_z + (d(t * pp - t_s)\psi/c)m$
      end while
      if $|\mathcal{N}| \geq 10$ **then**
        $a \leftarrow z - 10$
        $b \leftarrow |\mathcal{N}| - z + 1$
        $\psi \leftarrow \alpha^a(1 - \alpha^b)/((1 - \alpha)\delta_{10})$
        $\mathcal{G}_{10} \leftarrow \mathcal{G}_{10} + (d(t * pp - t_s)\psi/c)m$
        $\psi \leftarrow \log(\delta_{10})(a\alpha^{a-1} - (a+b)\alpha^{a+b-1})/(1-\alpha) + (\alpha^a - \alpha^{a+b})/(1-\alpha)^2$
        $\mathcal{G}_{11} \leftarrow \mathcal{G}_{11} + (d(t * pp - t_s)\psi/c)m$
      end if
    end if
  end if
end function

5 Conclusion

References

Bartlett, N., Pfau, D., and Wood, F. (2010). Forgetting counts : Constant memory inference for a dependent hierarchical Pitman-Yor process. In *(*to appear) ICML.

Cleary, J. G. and Teahan, W. J. (1997). Unbounded length contexts for PPM. *The Computer Journal*, 40:67–75.

Gasthaus, J., Wood, F., and Teh, Y. W. (2010). Lossless compression based on the Sequence Memoizer. In *Data Compression Conference 2010*, pages 337–345.

Mahoney, M. V. (2005). Adaptive weighing of context models for lossless data compression. Technical report, Florida Tech. Technical Report CS-2005-16, 2005.

Peltola, H. and Tarhio, J. (2002). URL: `http://cs.hut.fi/u/tarhio/ppmz`.

Willems, F. M. J. (1998). The context-tree weighting method: Extensions. *IEEE Transactions on Information Theory*, 44(2):792–798.

Willems, F. M. J. (2009). CTW website. URL: `http://www.ele.tue.nl/ctw/`.

Wood, F., Archambeau, C., Gasthaus, J., James, L., and Teh, Y. W. (2009). A stochastic memoizer for sequence data. In *Proceedings of the 26th International Conference on Machine Learning*, pages 1129–1136, Montreal, Canada.