Efficient Logistic Regression with Stochastic Gradient Descent

William Cohen
A FEW MORE COMMENTS ON SPARSIFYING THE LOGISTIC REGRESSION UPDATE
Learning as optimization for regularized logistic regression

• Final algorithm:

\[ w^j = w^j + \lambda(y - p)x^j - \lambda 2\mu w^j \]

• Initialize hashtable \( W \)

• For each iteration \( t = 1, \ldots, T \)
  
  – For each example \( (x_i, y_i) \)

  • \( p_i = \ldots \)

  • For each feature \( W[j] \)

    – \( W[j] = W[j] - \lambda 2\mu W[j] \)

    – If \( x_i^j > 0 \) then

      » \( W[j] = W[j] + \lambda (y_i - p^i)x_j \)
Learning as optimization for regularized logistic regression

- Final algorithm:
  \[ w^j = w^j + \lambda(y - p)x^j - \lambda 2 \mu w^j \]

- Initialize hashtable \( W \)

- For each iteration \( t=1,...,T \)
  - For each example \( (x_i,y_i) \)
    - \( p_i = \ldots \)
    - For each feature \( W[j] \)
      - \( W[j] *= (1 - \lambda 2 \mu) \)
      - If \( x_i^j > 0 \) then
        \[ W[j] = W[j] + \lambda(y_i - p^i)x_j \]
Learning as optimization for regularized logistic regression

• Final algorithm: 
  \[ w^j = w^j + \lambda(y - p)x^j - \lambda 2\mu w^j \]

• Initialize hashtable \( W \)

• For each iteration \( t=1,\ldots,T \)
  – For each example \( (x_i, y_i) \)
    • \( p_i = \ldots \)
    • For each feature \( W[j] \)
      – If \( x_i^j > 0 \) then
        \[ W[j] *= (1 - \lambda 2\mu)^A \]
        \[ W[j] = W[j] + \lambda(y_i - p^i)x_j \]

A is number of examples seen since the last time we did an \( x>0 \) update on \( W[j] \)
Learning as optimization for regularized logistic regression

- Final algorithm:
  \[ w^j = w^j + \lambda(y - p)x^j - \lambda 2\mu w^j \]

- Initialize hashtables \( W, A \) and set \( k=0 \)

- For each iteration \( t=1,...,T \)
  - For each example \( (x_i, y_i) \)
    - \( p_i = ... ; k++ \)
    - For each feature \( W[j] \)
      - If \( x_i^j > 0 \) then
        » \( W[j] = (1 - \lambda 2\mu)^{k-A[j]} \)
        » \( W[j] = W[j] + \lambda(y_i - p^i)x_j \)
        » \( A[j] = k \)

\( k-A[j] \) is number of examples seen since the last time we did an \( x>0 \) update on \( W[j] \)
Comments

• Same trick can be applied in other contexts
  – Other regularizers (eg L1, ...)
  – Conjugate gradient (Langford)
  – FTRL (Follow the regularized leader)
  – Voted perceptron averaging
  – ...?
SPARSIFYING THE AVERAGED PERCEPTRON UPDATE
Complexity of perceptron learning

- **Algorithm:** $O(n)$
- $v=0$
- for each example $x, y$:
  - if $\text{sign}(v.x) \neq y$
    - $v = v + yx$  \( O(|x|)=O(|d|) \)
  - for $x_i \neq 0$, $v_i += yx_i$
- init hashtable
Complexity of *averaged* perceptron

- Algorithm: \( \mathcal{O}(\eta) \) \( \mathcal{O}(n|V|) \)
- \( vk=0 \)
- \( va = 0 \)
- for each example \( x,y \):
  - if \( \text{sign}(vk.x) \neq y \) \( \mathcal{O}(|V|) \)
    - \( va = va + vk \)
    - \( vk = vk + yx \)
    - \( mk = 1 \) \( \mathcal{O}(|x|) = \mathcal{O}(|d|) \)
  - else
    - \( nk++ \)
- init hashtables
- for \( vk_i \neq 0 \), \( va_i += vk_i \)
- for \( x_i \neq 0 \), \( v_i += yx_i \)
Alternative averaged perceptron

• Algorithm:
  • $v_k = 0$
  • $v_a = 0$
  • for each example $x, y$:
    – $v_a = v_a + v_k$
    – $t = t + 1$
    – if sign($v_k \cdot x$) $\neq y$
      • $v_k = v_k + y^*x$
  • Return $v_a/t$

Observe:

$v_k = \sum_{j \in S_k} y_j x_j$

$S_k$ is the set of examples including the first $k$ mistakes
Alternative averaged perceptron

- Algorithm:
  - $v_k = 0$
  - $v_a = 0$
  - for each example $x,y$:
    - $v_a = v_a + \sum_{j \in S_k} y_j x_j$
    - $t = t+1$
    - if $\text{sign}(v_k.x) \neq y$
      - $v_k = v_k + y^*x$
  - Return $v_a/t$

So when there’s a mistake at time $t$ on $x,y$:

$y^*x$ is added to $v_a$ on every subsequent iteration

Suppose you know $T$, the total number of examples in the stream…
Alternative averaged perceptron

• Algorithm:
  • $\mathbf{v}_k = 0$
  • $\mathbf{v}_a = 0$
  • for each example $\mathbf{x}, y$:
    • $\mathbf{v}_a = \mathbf{v}_a + \sum_{j \in S_k} y_j \mathbf{x}_j$
    • $t = t + 1$
    • if $\text{sign}(\mathbf{v}_k \cdot \mathbf{x}) \neq y$
      • $\mathbf{v}_k = \mathbf{v}_k + y^* \mathbf{x}$
      • $\mathbf{v}_a = \mathbf{v}_a + (T - t) y^* \mathbf{x}$
  • Return $\mathbf{v}_a / T$

T = the total number of examples in the stream...

Unpublished? I figured this out recently, Leon Bottou knows it too
Formalization of the “Hash Trick”:

First: Review of Kernels
The kernel perceptron

Mathematically the same as before … but allows use of the kernel trick
The kernel perceptron

Compute: \( y_i = v_k \cdot x_i \)

If mistake: \( v_{k+1} = v_k + y_i x_i \)

\[ \hat{y} = \sum_{x_{k+} \in FN} K(x_i, x_{k+}) - \sum_{x_{k-} \in FP} K(x_i, x_{k-}) \]

Mathematically the same as before … but allows use of the “kernel trick”

Other kernel methods (SVM, Gaussian processes) aren’t constrained to limited set (+1/-1/0) of weights on the \( K(x,v) \) values.
Some common kernels

• Linear kernel:
  \[ K(x, x') \equiv x \cdot x' \]

• Polynomial kernel:
  \[ K(x, x') \equiv (x \cdot x' + 1)^d \]

• Gaussian kernel:
  \[ K(x, x') \equiv e^{-\frac{\|x-x'\|^2}{\sigma}} \]

• More later…. 
Kernels 101

• Duality
  – and computational properties
  – Reproducing Kernel Hilbert Space (RKHS)
• Gram matrix
• Positive semi-definite
• Closure properties
Explicitly map from $x$ to $\phi(x)$ – i.e. to the point corresponding to $x$ in the Hilbert space

Implicitly map from $x$ to $\phi(x)$ by changing the kernel function $K$

- **Duality**: two ways to look at this

\[
\hat{y} = x \cdot w = K(x, w)
\]
\[
w = \sum_{x_{k^+} \in FN} x_{k^+} - \sum_{x_{k^-} \in FP} x_{k^-}
\]

\[
\hat{y} = \sum_{x_{k^+} \in FN} K(x_i, x_{k^+}) - \sum_{x_{k^-} \in FP} K(x_i, x_{k^-})
\]
\[
K(x, x_k) \equiv \phi(x) \cdot \phi(x_k)
\]

\[
\hat{y} = \sum_{x_{k^+} \in FN} K(x_i, x_{k^+}) - \sum_{x_{k^-} \in FP} K(x_i, x_{k^-})
\]
\[
K(x, x_k) \equiv \phi(x') \cdot \phi(x_k')
\]

Two different computational ways of getting the same behavior
Kernels 101

• Duality
• Gram matrix: \( k_{ij} = K(x_i, x_j) \)

\[ K(x, x') = K(x', x) \Rightarrow \text{Gram matrix is symmetric} \]

\[ K(x, x) > 0 \Rightarrow \text{diagonal of } K \text{ is positive} \Rightarrow K \text{ is “positive semi-definite”} \Rightarrow z^T K z \geq 0 \text{ for all } z \]
Kernels 101

• Duality
• Gram matrix: $K: k_{ij} = K(x_i, x_j)$

$K(x, x') = K(x', x) \Rightarrow$ Gram matrix is *symmetric*

$K(x, x) > 0 \Rightarrow$ diagonal of $K$ is positive $\iff$ $K$ is “positive semi-definite” $\iff$ … $\iff$ $z^T K z \geq 0$ for all $z$

**Fun fact:** Gram matrix positive semi-definite $\iff$

$K(x_i, x_j) = \phi(x_i), \phi(x_j)$ for some $\phi$

**Proof:** $\phi(x)$ uses the eigenvectors of $K$ to represent $x$
HASH KERNELS AND “THE HASH TRICK”
Question

• Most of the weights in a classifier are
  — small and not important
Hash Kernels

Qinfeng Shi, James Petterson
Australian National University and NICTA,
Canberra, Australia

John Langford, Alex Smola, Alex Strehl
Yahoo! Research
New York, NY and Santa Clara, CA, USA

Gideon Dror
Department of Computer Science
Academic College of Tel-Aviv-Yaffo, Israel

Vishy Vishwanathan
Department of Statistics
Purdue University, IN, USA
Some details

Slightly different hash to avoid systematic bias

\[ V[h] = \sum_{j:hash(j) \% R = h} x_i^j \]

\[ \varphi[h] = \sum_{j:hash(j) \% m = h} \xi(j)x_i^j, \text{ where } \xi(j) \in \{-1, +1\} \]

\( m \) is the number of buckets you hash into (\( R \) in my discussion)
Some details

Slightly different hash to avoid systematic bias

$$\varphi[h] = \sum_{j: \text{hash}(j) \% m = h} \xi(j)x_i^j,$$

where $$\xi(j) \in \{-1,+1\}$$

**Lemma 2** The hash kernel is unbiased, that is

$$E_\phi[\langle x, x' \rangle_\phi] = \langle x, x' \rangle.$$ Moreover, the variance is

$$\sigma^2_{x,x'} = \frac{1}{m} \left( \sum_{i \neq j} x_i^2 x_j'^2 + x_i x_i' x_j x_j' \right),$$

and thus, for $$\|x\|_2 = \|x'\|_2 = 1$$,

$$\sigma^2_{x,x'} = O \left( \frac{1}{m} \right).$$

$$m$$ is the number of buckets you hash into ($$R$$ in my discussion)
**Some details**

**Theorem 3** Let $\epsilon < 1$ be a fixed constant and $x$ be a given instance. Let $\eta = \frac{\|x\|_{\infty}}{\|x\|_2}$. Under the assumptions above, the hash kernel satisfies the following inequality

$$\Pr \left\{ \frac{\|x\|_\phi^2 - \|x\|_2^2}{\|x\|_2^2} \geq \sqrt{2}\sigma_{x,x} + \epsilon \right\} \leq \exp\left(-\frac{\sqrt{\epsilon}}{4\eta}\right).$$

I.e. – a hashed vector is probably close to the original vector.
Some details

Corollary 4 For two vectors $x$ and $x'$, let us define

$$
\sigma := \max(\sigma_{x,x}, \sigma_{x',x'}, \sigma_{x-x',x-x'})
$$

$$
\eta := \min \left( \frac{\|x\|_\infty}{\|x\|_2}, \frac{\|x'\|_\infty}{\|x'\|_2}, \frac{\|x-x'\|_\infty}{\|x-x'\|_2} \right).
$$

Also let $\Delta = \|x\|^2 + \|x'\|^2 + \|x-x'\|^2$. Under the assumptions above, we have that

$$
\Pr \left[ |\langle x, x' \rangle_\phi - \langle x, x' \rangle| > (\sqrt{2}\sigma + \epsilon)\Delta/2 \right] < 3e^{-\frac{\sqrt{\epsilon}}{4\eta}}.
$$

I.e. the inner products between $x$ and $x'$ are probably not changed too much by the hash function: a classifier will probably still work
Corollary 5  Denote by $X = \{x_1, \ldots, x_n\}$ a set of vectors which satisfy $\|x_i - x_j\|_\infty \leq \eta \|x_i - x_j\|_2$ for all pairs $i, j$. In this case with probability $1 - \delta$ we have for all $i, j$

$$\left| \frac{\|x_i - x_j\|_\phi^2 - \|x_i - x_j\|_2^2}{\|x_i - x_j\|_2^2} \right| \leq \sqrt{\frac{2}{m}} + 64\eta^2 \log^2 \frac{n}{2\delta}.$$

This means that the number of observations $n$ (or correspondingly the size of the un-hashed kernel matrix) only enters logarithmically in the analysis.
The hash kernel: implementation

• One problem: debugging is harder
  – Features are no longer meaningful
  – There’s a new way to ruin a classifier
    • Change the hash function 😞
• You can separately compute the set of all words that hash to $h$ and guess what features mean
  – Build an inverted index $h \rightarrow w_1, w_2, ...,$
A variant of feature hashing

• Hash each feature *multiple times* with different hash functions
• Now, each $w$ has $k$ chances to *not* collide with another useful $w'$
• An easy way to get multiple hash functions
  – Generate some random strings $s_1, \ldots, s_L$
  – Let the $k$-th hash function for $w$ be the ordinary hash of concatenation $w \cdot s_k$

$$V[h] = \sum_k \sum_{j: \text{hash}(j \cdot s_k) \% R = h} x_i^j$$
A variant of feature hashing

• Why would this work?

\[ V[h] = \sum \sum x_{ij} \]

\[ k \quad j:hash(j \cdot s_k) \% R = h \]

• Claim: with 100,000 features and 100,000,000 buckets:
  - \( k=1 \) \( \Rightarrow \) \( \Pr(\text{any duplication}) \approx 1 \)
  - \( k=2 \) \( \Rightarrow \) \( \Pr(\text{any duplication}) \approx 0.4 \)
  - \( k=3 \) \( \Rightarrow \) \( \Pr(\text{any duplication}) \approx 0.01 \)
Experiments

- RCV1 dataset
- Use hash kernels with TF-approx-IDF representation
  - TF and IDF done at level of hashed features, not words
  - IDF is approximated from *subset* of data

Shi et al, JMLR 2009
## Results

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Pre</th>
<th>TrainTest</th>
<th>Error %</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSGD</td>
<td>303.60s</td>
<td>10.38s</td>
<td>6.02</td>
</tr>
<tr>
<td>VW</td>
<td>303.60s</td>
<td>87.63s</td>
<td>5.39</td>
</tr>
<tr>
<td>VWC</td>
<td>303.60s</td>
<td>5.15s</td>
<td>5.39</td>
</tr>
<tr>
<td>HK</td>
<td>0s</td>
<td>25.16s</td>
<td>5.60</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dim</th>
<th>#Unique</th>
<th>Collision %</th>
<th>Error %</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{24}$</td>
<td>285614</td>
<td>0.82</td>
<td>5.586</td>
</tr>
<tr>
<td>$2^{22}$</td>
<td>278238</td>
<td>3.38</td>
<td>5.655</td>
</tr>
<tr>
<td>$2^{20}$</td>
<td>251910</td>
<td>12.52</td>
<td>5.594</td>
</tr>
<tr>
<td>$2^{18}$</td>
<td>174776</td>
<td>39.31</td>
<td>5.655</td>
</tr>
<tr>
<td>$2^{16}$</td>
<td>64758</td>
<td>77.51</td>
<td>5.763</td>
</tr>
<tr>
<td>$2^{14}$</td>
<td>16383</td>
<td>94.31</td>
<td>6.096</td>
</tr>
<tr>
<td>Data Sets</td>
<td>#Train</td>
<td>#Test</td>
<td>#Labels</td>
</tr>
<tr>
<td>-----------</td>
<td>--------</td>
<td>-------</td>
<td>---------</td>
</tr>
<tr>
<td>RCV1</td>
<td>781,265</td>
<td>23,149</td>
<td>2</td>
</tr>
<tr>
<td>Dmoz L2</td>
<td>4,466,703</td>
<td>138,146</td>
<td>575</td>
</tr>
<tr>
<td>Dmoz L3</td>
<td>4,460,273</td>
<td>137,924</td>
<td>7,100</td>
</tr>
</tbody>
</table>

Table 1: Text data sets. #X denotes the number of observations in X.

<table>
<thead>
<tr>
<th></th>
<th>HLF ($2^{28}$)</th>
<th>HLF ($2^{24}$)</th>
<th>HF</th>
<th>no hash</th>
<th>U base</th>
<th>P base</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>error</td>
<td>error</td>
<td>error</td>
<td>mem</td>
<td>mem</td>
<td>mem</td>
</tr>
<tr>
<td>L2</td>
<td>30.12</td>
<td>30.71</td>
<td>31.28</td>
<td>2.25G ($2^{19}$)</td>
<td>7.85G</td>
<td>99.83</td>
</tr>
<tr>
<td>L3</td>
<td>52.10</td>
<td>53.36</td>
<td>51.47</td>
<td>1.73G ($2^{15}$)</td>
<td>96.95G</td>
<td>99.99</td>
</tr>
</tbody>
</table>

Table 5: Misclassification and memory footprint of hashing and baseline methods on DMOZ. HLF: joint hashing of labels and features. HF: hash features only. no hash: direct model (not implemented as too large, hence only memory estimates—we have 1,832,704 unique words). U base: baseline of uniform classifier. P base: baseline of majority vote. mem: memory used for the model. Note: the memory footprint in HLF is essentially independent of the number of classes used.
Figure 2: Test accuracy comparison of KNN and Kmeans on Dmoz with various sample sizes. Left: results on L2. Right: results on L3. Hash kernel (2^{28}) result is used as an upper bound.