Efficient Logistic Regression with Stochastic Gradient Descent

William Cohen

SGD FOR LOGISTIC REGRESSION

SGD for Logistic regression

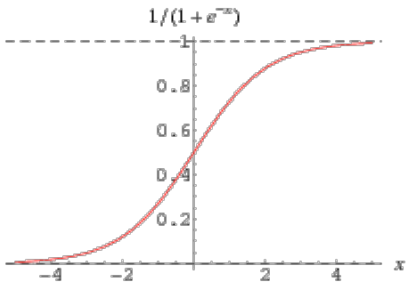
- Start with Rocchio-like linear classifier:

$$\hat{y} = sign(\mathbf{x} \cdot \mathbf{w})$$

- Replace sign(...) with something differentiable:
  - Also scale from 0-1 not -1 to +1

$$\hat{y} = \sigma(\mathbf{x} \cdot \mathbf{w}) = p$$

$$\sigma(s) = \frac{1}{1 + e^{-s}}$$



- Decide to optimize:

$$LCL(y \mid \mathbf{w}, \mathbf{x}) = \begin{cases} \log \sigma(\mathbf{w} \cdot \mathbf{x}) & y = 1 \\ \log(1 - \sigma(\mathbf{w} \cdot \mathbf{x})) & y = 0 \end{cases}$$

- Differentiate….

$$= \log\left(\sigma(\mathbf{w} \cdot \mathbf{x})^{y} (1 - \sigma(\mathbf{w} \cdot \mathbf{x}))^{1-y}\right)$$

$$\log P(Y = y|X = \mathbf{x}, \mathbf{w}) = \begin{cases} \log p & \text{if } y = 1 \\ \log(1-p) & \text{if } y = 0 \end{cases}$$

$$p = \sigma(\mathbf{x} \cdot \mathbf{w})$$

Magically, when we differentiate, we end up with something very simple and elegant…..

$$\frac{\partial}{\partial \mathbf{w}} L(\mathbf{w} \mid y, \mathbf{x}) = (y - p)\mathbf{x}$$

$$\frac{\partial}{\partial w^j} L(\mathbf{w} \mid y, \mathbf{x}) = (y - p)x^j$$

The update for gradient descent with rate λ is just:

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + \lambda(y - p)\mathbf{x}$$

An observation: sparsity!

$$\frac{\partial}{\partial w^j} \log P(Y = y | X = \mathbf{x}, \mathbf{w}) = (y - p)x^j$$

Key computational point:

- if $x^j = 0$ then the gradient of w^j is zero
- so when processing an example you only need to update weights for the **non-zero** features of an example.

SGD for logistic regression

- The algorithm: $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + \lambda(y - p)\mathbf{x}$

1. Initialize a hashtable W
2. For $t = 1, \ldots, T$
   - For each example $\mathbf{x}_i, y_i$: *-- do this in random order*
     - Compute the prediction for $\mathbf{x}_i$:

$$p_i = \frac{1}{1 + \exp(-\sum_{j:x_i^j>0} x_i^j w^j)}$$

     - For each non-zero feature of $\mathbf{x_i}$ with index j and value x^j:
       * If j is not in W, set $W[j] = 0$.
       * Set $W[j] = W[j] + \lambda(y_i - p_i)x^j$
3. Output the hash table W.

Adding regularization

- Replace LCL

$$\log P(Y = y|X = \mathbf{x}, \mathbf{w}) = \begin{cases} \log p & \text{if } y = 1 \\ \log(1-p) & \text{if } y = 0 \end{cases}$$

- with LCL + penalty for large weights, eg

$$LCL - \mu \sum_{j=1}^{d} (w^j)^2$$

- So:

$$\frac{\partial}{\partial w^j} \log P(Y = y|X = \mathbf{x}, \mathbf{w}) = (y - p)x^j$$

- becomes:

$$\frac{\partial}{\partial w^j} \log P(Y = y|X = \mathbf{x}, \mathbf{w}) - \mu \sum_{j=1}^{d} (w^j)^2 = (y - p)x^j - 2\mu w^j$$

Regularized logistic regression

- Replace LCL

$$\log P(Y = y|X = \mathbf{x}, \mathbf{w}) = \begin{cases} \log p & \text{if } y = 1 \\ \log(1-p) & \text{if } y = 0 \end{cases}$$

- with LCL + penalty for large weights, eg

$$LCL - \mu \sum_{j=1}^{d} (w^j)^2$$

- So the update for wj becomes:

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

- Or

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

Naively this is not a sparse update

- Algorithm:

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

1. Initialize a hashtable W
2. For $t = 1, \ldots, T$
   - For each example $\mathbf{x}_i, y_i$:
     - Compute the prediction for $\mathbf{x}_i$:

$$p_i = \frac{1}{1 + \exp(-\sum_{j:x_i^j>0} x_i^j w^j)}$$

     - For each ~~non-zero~~ feature of $\mathbf{x_i}$ with index j and value x^j:
       - If j is not in W, set $W[j] = 0$.
       - Set $W[j] = W[j] + \lambda(y - p)x^j - \lambda 2\mu w^j$
3. Output the hash table W.

Time goes from O(nT) to O(mVT) where

- n = number of non-zero entries,
- m = number of examples
- V = number of features
- T = number of passes over data

Sparse 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 ($\mathbf{x}_i$,y_i)
    - p_i = … ; *k++*
    - For each non-zero feature *W[j]*
      - *W[j]* *= $(1 - \lambda 2\mu)^{k-A[j]}$
      - *W[j]* = *W[j]* + $\lambda(y_i - p^i)x_j$
      - *A[j]* = *k*

- k = "clock" reading
- A[j] = clock reading last time feature j was "active"
- we implement the "weight decay" update using a "lazy" strategy: weights are decayed in one shot when a feature is "active"

Sparse regularized logistic regression (v2)

- Initialize hashtables W, A and set $k=0$
- For each iteration t=1,…T
  - For each example $(\mathbf{x}_i, y_i)$
    - k++
    - For each non-zero feature $W[j]$
      - $W[j] \mathrel{*}= (1 - \lambda 2\mu)^{k-A[j]}$ **
    - $p_i = \ldots$
    - For each non-zero feature $W[j]$
      - $W[j] = W[j] + \lambda(y_i - p^i)x_j$
      - $A[j] = k$
- Finally:
  - Before you write out each W[j] do **

- k = "clock" reading
- A[j] = clock reading last time feature j was "active"
- we implement the "weight decay" update using a "lazy" strategy: weights are decayed in one shot when a feature is "active"

Summary

- What's happened here:
  - Our update involves a *sparse part* and a *dense part*
    - Sparse: empirical loss on this example
    - Dense: regularization loss – not affected by the example
  - We remove the *dense part* of the update
    - Old example update:
      - for each feature { do something example-independent}
      - For each active feature { do something example-dependent}
    - New example update:
      - For each active feature :
        - » {simulate the prior example-independent updates}
        - » {do something example-dependent}

Summary

- We can separate the LCL update and weight decay updates but remember:
  - we need to apply "weight decay" to *just the active features in an example* x_i before we compute the prediction p_i
  - we need to apply "weight decay" to all features before we save the classifier
  - my suggestion:
    - an abstraction for a logistic regression classifier

A possible SGD implementation

```
class SGDLogistic Regression {
   /** Predict using current weights **/
     double predict(Map features);
   /** Apply weight decay to a single feature and record when in A[ ]**/
     void regularize(string feature, int currentK);
     /** Regularize all features then save to disk **/
   void save(string fileName,int currentK);
   /** Load a saved classifier **/
   static SGDClassifier load(String fileName);
     /** Train on one example **/
   void train1(Map features, double trueLabel, int k) {
         // regularize each feature
        // predict and apply update
  }
}
// main 'train' program assumes a stream of randomly-ordered examples and
     outputs classifier to disk; main 'test' program prints predictions for each
     test case in input.
```

A possible SGD implementation

class SGDLogistic Regression {

    …

}

// main 'train' program assumes a stream of **randomly-ordered** examples and outputs classifier to disk; main 'test' program prints predictions for each test case in input.

    <100 lines (in python)

Other mains:

- A "shuffler:"
  - stream thru a training file T times and output instances
  - output is randomly ordered, as much as possible, given a buffer of size B
- Something to collect predictions + true labels and produce error rates, etc.

A possible SGD implementation

- Parameter settings:
  - $W[j] \; *= (1 - \lambda 2\boldsymbol{\mu})^{k-A[j]}$
  - $W[j] = W[j] + \boldsymbol{\lambda}(y_i - p^i)x_j$

- I didn't tune especially but used
  - $\mu = 0.1$
  - $\lambda = \eta * E^{-2}$ where E is "epoch", $\eta = ½$
    - epoch: number of times you've iterated over the dataset, starting at E=1

BOUNDED-MEMORY LOGISTIC REGRESSION

Outline

- Logistic regression and SGD
  - Learning as optimization
  - Logistic regression:
    - a linear classifier optimizing P(y|**x**)
  - Stochastic gradient descent
    - "streaming optimization" for ML problems
  - Regularized logistic regression
  - Sparse regularized logistic regression
  - **Memory-saving logistic regression**

Question

- In text classification most words are
  a. rare
  b. not correlated with any class
  c. given low weights in the LR classifier
  d. unlikely to affect classification
  e. not very interesting

Question

- In text classification most bigrams are
  a. rare
  b. not correlated with any class
  c. given low weights in the LR classifier
  d. unlikely to affect classification
  e. not very interesting

Question

- Most of the weights in a classifier are
  - important
  - not important

How can we exploit this?

- One idea: combine uncommon words together *randomly*
- Examples:
  - replace all occurrances of "humanitarianism" or "biopsy" with "humanitarianismOrBiopsy"
  - replace all occurrances of "schizoid" or "duchy" with "schizoidOrDuchy"
  - replace all occurrances of "gynecologist" or "constrictor" with "gynecologistOrConstrictor"
  - …
- For Naïve Bayes this breaks independence assumptions
  - it's not obviously a problem for logistic regression, though
- I could combine
  - two low-weight words (won't matter much)
  - a low-weight and a high-weight word (won't matter much)
  - two high-weight words (not very likely to happen)
- How much of this can I get away with?
  - certainly a little
  - is it enough to make a difference? how much memory does it save?

How can we exploit this?

- Another observation:
  - the values in my hash table are *weights*
  - the keys in my hash table are *strings* for the feature names
    - We need them to avoid collisions

- But maybe we don't care about collisions?
  - Allowing "schizoid" & "duchy" to collide is equivalent to replacing all occurrences of "schizoid" or "duchy" with "schizoidOrDuchy"

Learning as optimization for regularized logistic regression

- 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 ($\mathbf{x}_i$,y_i)
    - $p_i = \ldots$; *k++*
    - For each feature *j:* $x_i^j > 0$:
      - » *W[j]* *= $(1 - \lambda 2\mu)^{k-A[j]}$
      - » $W[j] = W[j] + \lambda(y_i - p^i)x_j$
      - » $A[j] = k$

Learning as optimization for regularized logistic regression

- Algorithm: $w^j = w^j + \lambda(y - p)x^j - \lambda 2\mu w^j$
- Initialize arrays *W, A* of size *R* and set *k=0*
- For each iteration t=1,…T
  - For each example ($\mathbf{x}_i$,y_i)
    - Let V be hash table so that $V[h] = \sum_{j:hash(x_i^j)\%R=h} x_i^j$
    - $p_i = \ldots$; *k++*
    - For each hash value *h: V[h]*>0:
      - » $W[h] \mathrel{*}= (1 - \lambda 2\mu)^{k-A[j]}$
      - » $W[h] = W[h] + \lambda(y_i - p^i)V[h]$
      - » $A[j] = k$

Learning as optimization for regularized logistic regression

- Algorithm: $w^j = w^j + \lambda(y - p)x^j - \lambda 2\mu w^j$
- Initialize arrays *W, A* of size *R* and set *k=0*
- For each iteration t=1,…T
  - For each example ($\mathbf{x}_i$,y_i)
    - Let V be hash table so that $V[h] = \sum_{j:hash(j)\%R==h} x_i^j$
    - ??? $p_i = \ldots$; *k++*

$$p \equiv \frac{1}{1 + e^{-\mathbf{x}\cdot\mathbf{w}}} \Rightarrow p \equiv \frac{1}{1 + e^{-\mathbf{V}\cdot\mathbf{w}}}$$

SOME EXPERIMENTS

Feature Hashing for Large Scale Multitask Learning

Kilian Weinberger KILIAN@YAHOO-INC.COM
Anirban Dasgupta ANIRBAN@YAHOO-INC.COM
John Langford JL@HUNCH.NET
Alex Smola ALEX@SMOLA.ORG
Josh Attenberg JOSH@CIS.POLY.EDU
Yahoo! Research, 2821 Mission College Blvd., Santa Clara, CA 95051 USA

ICML 2009

An interesting example

- Spam filtering for Yahoo mail
  - Lots of examples and lots of users
  - Two options:
    - one filter for everyone—but users disagree
    - one filter for each user—but some users are lazy and don't label anything
  - Third option:
    - classify *(msg,user)* pairs
    - features of message *i* are words $w_{i,1},\ldots,w_{i,ki}$
    - feature of user is his/her id *u*
    - features of **pair** are: $w_{i,1},\ldots,w_{i,ki}$ and $u \bullet w_{i,1},\ldots,u \bullet w_{i,ki}$
    - based on an idea by Hal Daumé

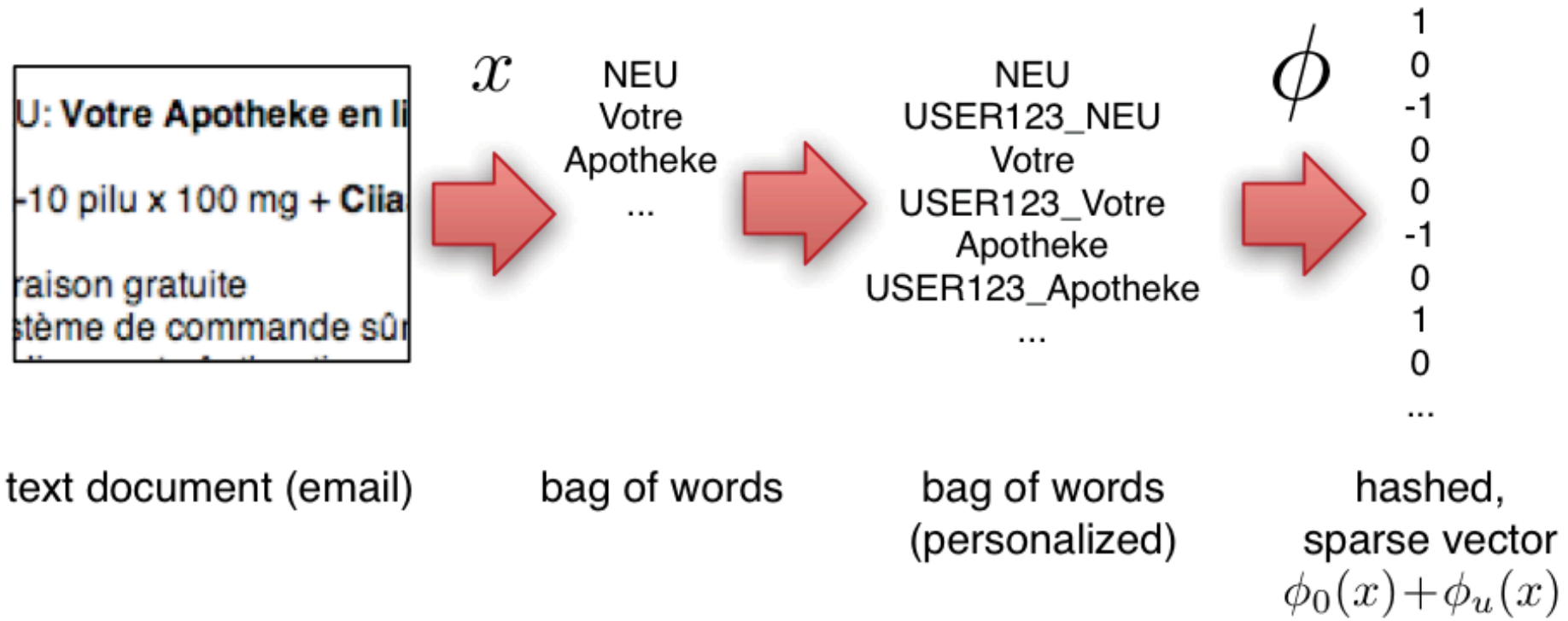
An example

- E.g., this email to wcohen

> Dear Madam/Sir,
>
> My name is Mohammed Azziz an investment Broker with SouthCoast Plc a company based in London United Kingdom our major activity is in the area of managing customers funds with targetted interest rates through provision and acquisition of loans to interested borrowers with the basic requisite. Our periodic checks on people and Companies located

- features:
  - dear, madam, sir,…. investment, broker,…, wcohen•dear, wcohen•madam, wcohen,…,
- idea: the learner will figure out how to personalize my spam filter by using the wcohen•X features

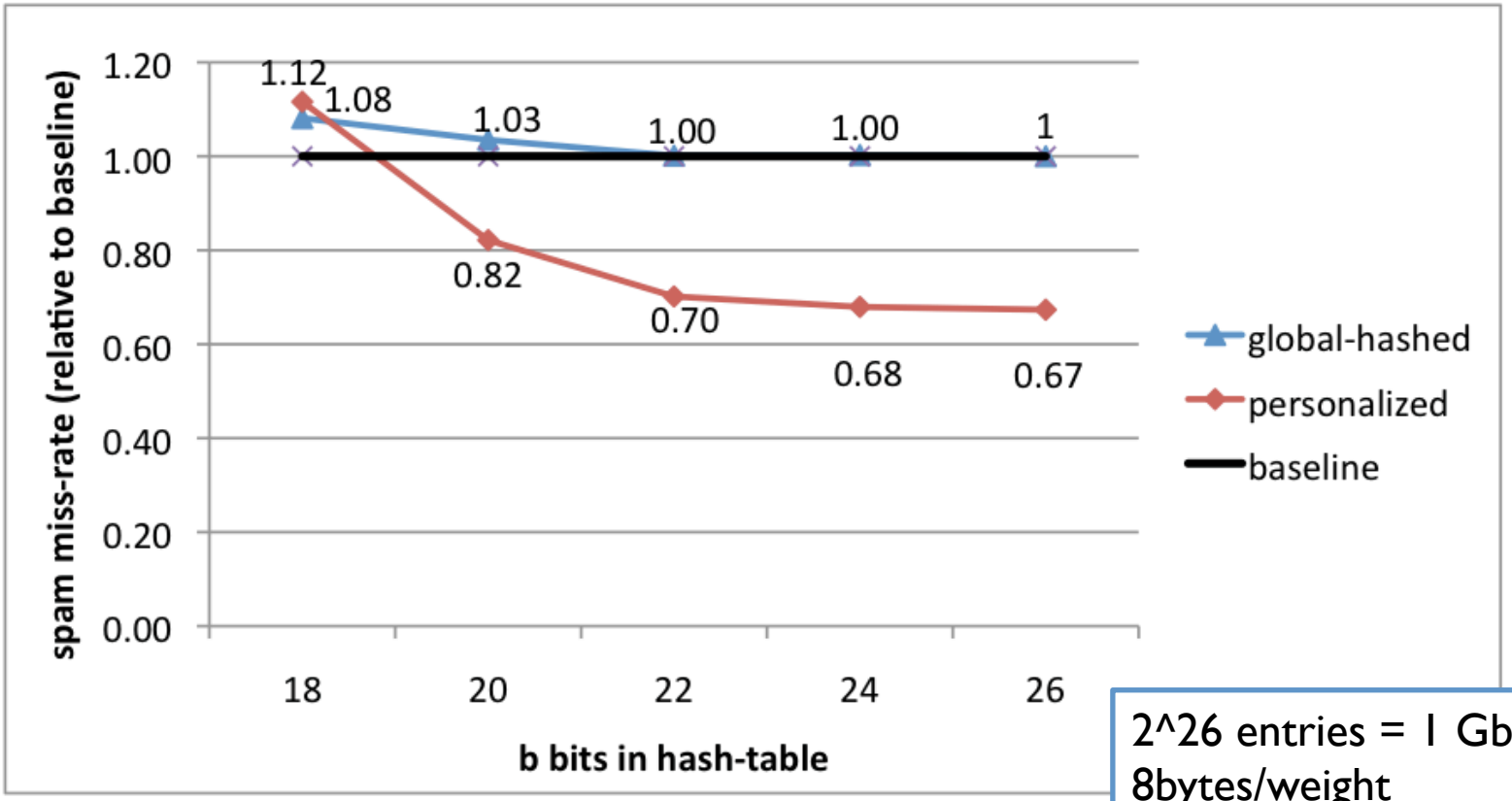
An example



Compute personalized features and multiple hashes on-the-fly:
a great opportunity to use several processors and speed up i/o

Experiments

- 3.2M emails
- 40M tokens
- 430k users
- 16T unique features – after personalization



2^26 entries = 1 Gb @ 8bytes/weight

Figure 2. The decrease of uncaught spam over the baseline classifier averaged over all users. The classification threshold was chosen to keep the not-spam misclassification fixed at 1%. The hashed global classifier (*global-hashed*) converges relatively soon, showing that the distortion error ϵ_d vanishes. The personalized classifier results in an average improvement of up to 30%.

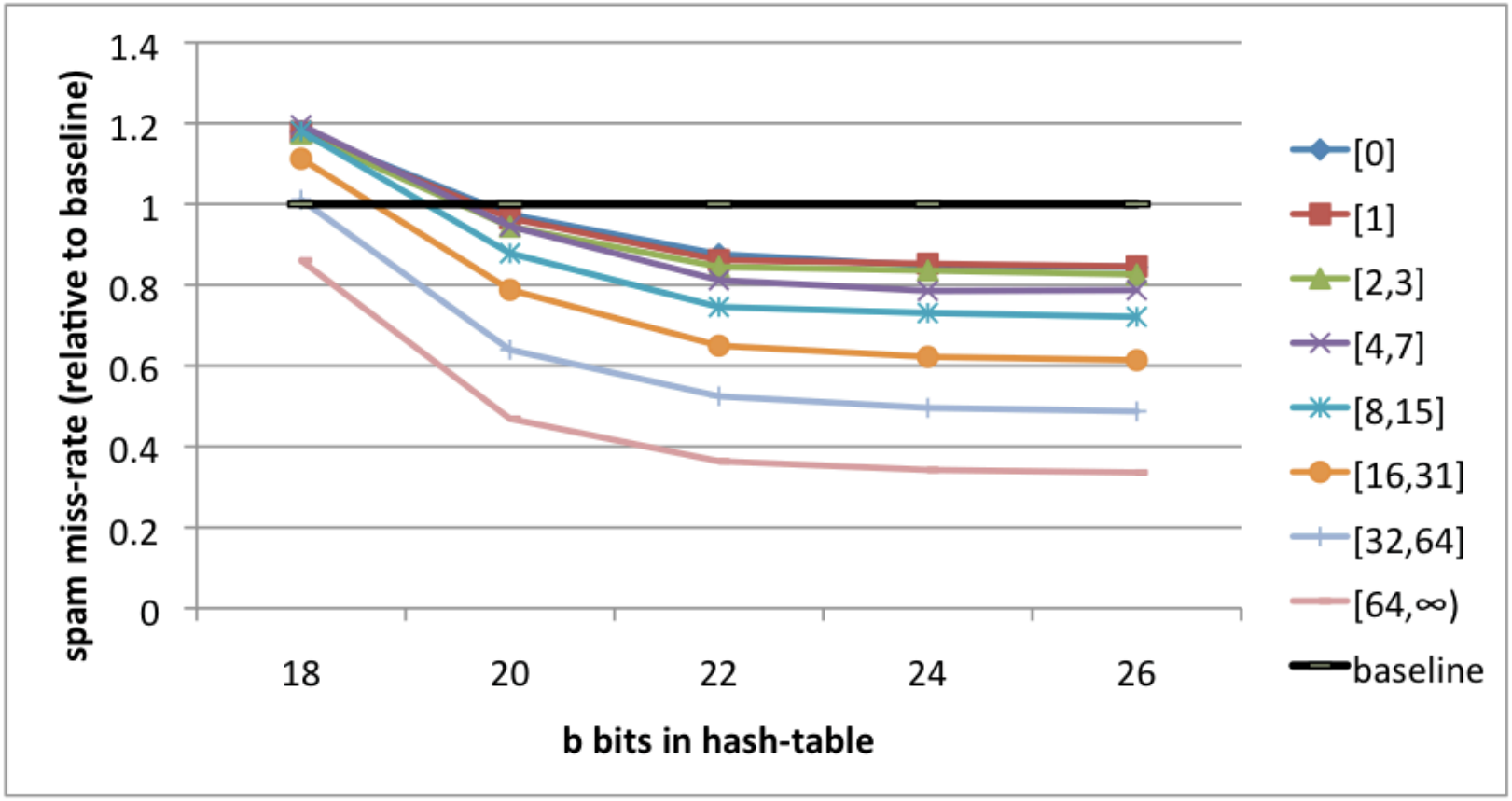


Figure 3. Results for users clustered by training emails. For example, the bucket $[8, 15]$ consists of all users with eight to fifteen training emails. Although users in buckets with large amounts of training data do benefit more from the personalized classifier (up-to 65% reduction in spam), even users that did not contribute to the training corpus at all obtain almost 20% spam-reduction.