Fundamental Techniques in Big Data

Data Streams, Trees, Learning, and Search

Ping Li
Outline

1. **Data streams, anomaly detection, and entropy estimation** (pages 3-13)

2. **Trees and boosting for accurate machine learning** (pages 14-77)

3. **Hashing algorithms for building large-scale and complex learning models**

4. **Hashing algorithms for indexing and efficient near neighbor search**
A data stream is a vector $A_t$ of length $D$, where $D = 2^{64}$ or even $D = 2^{128}$ is possible in network applications, e.g., (a pair of) IP addresses + port numbers.

At time $t$, there is an input stream $a_t = (i_t, I_t)$, $i_t \in [1, D]$ which updates $A_t$ according to

$$A_t[i_t] = A_{t-1}[i_t] + I_t$$

where $I_t$ is the increment/decrement of package size at $t$.

For network traffic, normally $A_t[i] \geq 0$, which is called the strict turnstile model and suffices for describing certain natural phenomena.

On the other hand, the general turnstile model (which allows $A_t[i] < 0$) is often used for comparing two streams, e.g., in network OD (origin-destination) flow analysis.
In data stream analysis, an important task is to compute the $\alpha$-th frequency moment $F(\alpha)$ and the Shannon entropy $H$:

$$F(\alpha) = \sum_{i=1}^{D} |A_t[i]|^\alpha, \quad H = -\sum_{i=1}^{D} \frac{|A_t[i]|}{F_1} \log \frac{|A_t[i]|}{F_1},$$

The exact computation of these summary statistics is not feasible because to do so one has to store the entire vector $A_t$ of length $D$, as the entries are time-varying.

Also, many applications (such as anomaly detections of network traffic) require computing the summary statistics in real-time.
Network Traffic and Data Streams

Network traffic is a typical example of high-rate data streams. An effective and reliable measurement of network traffic in real-time is crucial for anomaly detection and network diagnosis; and one such measurement metric is the Shannon entropy. The exact entropy measurement in real-time on high-speed links is however computationally prohibitive.

This plot is reproduced from a DARPA conference. One can view x-axis as the surrogate for time. Y-axis is the measured Shannon entropy, which exhibited a sudden sharp change at the time when an attack occurred.
Distributed Denial of Service (DDoS)

The DDoS attack is a representative example of network anomalies. A DDoS attack attempts to make computers unavailable to intended users, either by forcing users to reset the computers or by exhausting the resources of service-hosting sites. Hackers may maliciously saturate the victim machines by sending many external communication requests. DDoS attacks typically target sites such as banks, credit card payment gateways, or military sites.

A DDoS attack normally changes the statistical distribution of network traffic, which could be reliably captured by the abnormal variations in the measurements of Shannon entropy.

Apparently, the entropy measurements do not have to be “perfect” for detecting attacks. It is however crucial that the algorithms should be computationally efficient (i.e., real-time and one-pass) at low memory cost, because the traffic data generating by large high-speed networks are enormous and transient.
Stable Random Projections

Multiply the data stream vector \( A_t \in \mathbb{R}^D \) by a random matrix \( R \in \mathbb{R}^{D \times k} \), resulting in a vector \( X = A_t \times R \in \mathbb{R}^k \) with entries

\[
x_j = [A_t \times R]_j = \sum_{i=1}^{D} r_{ij} A_t[i], \quad j = 1, 2, \ldots, k
\]

where \( r_{ij} \sim S(\alpha, 1) \) is a \( \alpha \)-stable random variable with unit scale: \( E(e^{r_{ij} t}) = e^{-|t|^\alpha} \). The standard normal (or Cauchy) distribution is a special case with \( \alpha = 2 \) (or \( \alpha = 1 \)).

Entries of \( R \) on-demand using pseudo-random numbers. Thus, we only need to store \( X \in \mathbb{R}^k \). When a stream element \( a_t = (i_t, I_t) \) arrives, one updates the entries of \( X \):

\[
x_j \leftarrow x_j + I_t r_{i_t j}, \quad j = 1, 2, \ldots, k.
\]

By property of stable distributions, the samples \( x_j, j = 1 \) to \( k \), are also i.i.d. stable

\[
x_j = \sum_{i=1}^{D} r_{ij} A_t[i] \sim S \left( \alpha, F(\alpha) = \sum_{i=1}^{D} |A_t[i]|^\alpha \right)
\]
Entropy Estimation Using Frequency Moments

The Shannon entropy is essentially the derivative of the frequency moment at $\alpha = 1$. A common approach is to approximate the Shannon entropy by the Tsallis entropy

$$T_\alpha = \frac{1}{\alpha - 1} \left( 1 - \frac{F(\alpha)}{F(1)} \right), \quad \hat{T}_\alpha = \frac{1}{\alpha - 1} \left( 1 - \frac{\hat{F}(\alpha)}{\hat{F}(1)} \right).$$

which approaches the Shannon entropy $H$ as $\alpha \to 1$.

**Challenges:**

$$\text{Var} \left( T(\alpha) \right) \propto \frac{1}{(\alpha - 1)^2} = \frac{1}{\Delta^2}.$$ 

Theorists showed that one needs to choose $\Delta = |1 - \alpha| < 10^{-7}$, which means the number of samples has to be $O \left( 10^{14} \right)$.

In practice, one can exploit the bias-variance tradeoff but still needs to use an excessive number of samples, e.g., $10^6$. 
Our Breakthroughs

- For nonnegative data streams, we propose to use “maximally-skewed” stable random projections. This reduces the number of measurements to merely 10.

- For general data streams (for example, difference of two streams), we propose to make use of highly dependent estimates of frequency moments, which also substantially reduces the variance. The required number of measurements is merely 100.
Compressed Counting (CC) provides an ideal solution entropy estimation. For nonnegative data streams, i.e., $A_t[i] \geq 0$ at all times and all locations, the first moment is trivial:

$$F(1) = \sum_{i=1}^{D} |A_t[i]| = \sum_{i=1}^{D} A_t[i] = \sum_{s=0}^{t} I_s$$

where $I_s$ is the increment/decrement at time $s$. In other words, we just need a single counter to accumulate all the increments $I_s$. This observation lead to the conjecture that estimating $F(\alpha)$ should be also easy if $\alpha \approx 1$, which lead to the development of Compressed Counting which used maximally-skewed stable random projections.

$$\hat{T}_\alpha = \frac{1}{\alpha - 1} \left( 1 - \frac{\hat{F}(\alpha)}{F(1)^\alpha} \right)$$

$$Var \left( \hat{T}_\alpha \right) = \frac{1}{(\alpha - 1)^2} \frac{Var \left( \hat{F}(\alpha) \right)}{F(1)^{2\alpha}} \propto \frac{1}{F(1)^{2\alpha}}$$
Highly Dependent Moment Estimations for Entropy

\[ \hat{T}_\alpha = \frac{1}{\alpha - 1} \left( 1 - \frac{\hat{F}(\alpha)}{\hat{F}^{\alpha}(1)} \right). \]

If we make \( \hat{F}(\alpha) \) and \( \hat{F}^{\alpha}(1) \) highly dependent, we can also effectively reduce the variance of \( \frac{\hat{F}(\alpha)}{\hat{F}^{\alpha}(1)} \) to be essentially \( O \left( (1 - \alpha)^2 \right) \).

The actual estimator and especially its analysis are mathematically complicated (and counter-intuitive). The performance is excellent.
Conclusion for Entropy Estimation

• Practical data can often be viewed as streams. Anomaly detection by monitoring statistics of data streams is an effective approach. Entropy is a good choice of statistics.

• Entropy estimation however used to be a well-known difficult problem. $10^{14}$ (in theory) or $10^6$ (in practice) were needed in prior studies.

• Entropy estimation is now a trivial problem. For nonnegative data, Compressed Counting only needs 10 measurements for accurate estimates of entropy.

• For general data streams (e.g., difference of two streams), highly dependent stable projections needs only about 100 measurements.
References to Data Streams and Entropy Estimation


**Ref**: P. Li, *Improved Compressed Counting*, UAI 2009

**Ref**: P. Li and Cun-Hui Zhang *A New Algorithm for Compressed Counting with Applications in Shannon Entropy Estimation in Dynamic Data*, COLT 2012

**Ref**: P. Li and Cun-Hui Zhang *Entropy estimations using correlated symmetric stable random projections*, NIPS 2012
Recent Progress on Boosting and Trees

Why trees? Why are they not enough?

- Trees & boosting typically produce robust and fairly accurate learning results.
- Sometimes trees & boosting can be dramatically more accurate than other methods.
- Therefore, often in practice, trees & boosting should be the first learning tool to try.
- However, trees are not suitable for nominal categorical variables with many categories.
- Trees & boosting are often very slow and the model sizes are often large.
- Hashing + logistic regression (or DNN) is often a practical alternative.
References to Recent Progress on Boosting and Trees

Ref: P. Li, et. al., McRank: Learning to Rank with Multiple Classification and Gradient Boosting, NIPS 2007.

Ref: P. Li, ABC-Boost for Multi-Class Classification, ICML 2009 (arXiv 2008)

Ref: P. Li, Robust LogitBoost and ABC-LogitBoost, UAI 2010

Ref: P. Li, Learning to Rank Using Robust Logitboost, Yahoo! 2010 Learning to Rank Grand Challenges.
What is Classification?

An Example: USPS Handwritten Zipcode Recognition

Person 1:

Person 2:

Person 3:

The task: Teach the machine to automatically recognize the 10 digits.
Multi-Class Classification

Given a training data set

\[ \{ y_i, X_i \}_{i=1}^{N}, \quad X_i \in \mathbb{R}^p, \quad y_i \in \{0, 1, 2, \ldots, K - 1\} \]

the task is to learn a function to predict the class label \( y_i \) from \( X_i \).

- \( K = 2 \): binary classification
- \( K > 2 \): multi-class classification

Many important practical problems can be cast as (multi-class) classification.
For example, Li, Burges, and Wu, NIPS 2007
Learning to Ranking Using Multiple Classification and Gradient Boosting.
Logistic Regression for Classification

First learn the class probabilities

\[ \hat{p}_k = \Pr \{ y = k | X \}, \quad k = 0, 1, ..., K - 1, \]

\[ \sum_{k=0}^{K-1} \hat{p}_k = 1, \quad \text{(only } K - 1 \text{ degrees of freedom)}. \]

Then assign the class label according to

\[ \hat{y} | X = \arg\max_k \hat{p}_k \]
Multinomial Logit Probability Model

\[ p_k = \frac{e^{F_k}}{\sum_{s=0}^{K-1} e^{F_s}} \]

where \( F_k = F_k(x) \) is the function to be learned from the data.

**Classical logistic regression:**

\[ F(x) = \beta^T x \]

The task is to learn the coefficients \( \beta \).

**Flexible additive modeling:**

\[ F(x) = F^{(M)}(x) = \sum_{m=1}^{M} \rho_m h(x; a_m), \]
$h(x; a)$ is a pre-specified function (e.g., trees).

The task is to learn the parameters $\rho_m$ and $a_m$.

Both LogitBoost (Friedman et. al, 2000) and MART (Multiple Additive Regression Trees, Friedman 2001) adopted this model.
Need to find the best split point in each dimension and the best dimension (feature).

For regression, the prediction is averaged value of the points in the leaf node.
Regression Tree Split Criterion

Square error: $\sum_i (z_i - \bar{z})^2$  
Square error: $\sum_{x_i < s} (z_i - \bar{z}_L)^2 + \sum_{x_i \geq s} (z_i - \bar{z}_R)^2$

We can also use weighted square errors.
Seek $F_{i,k}$ to maximize the mutlinomial likelihood: Suppose $y_i = k$,

$$Lik \propto p_{i,0}^0 \times \ldots \times p_{i,k}^1 \times \ldots \times p_{i,K-1}^0 = p_{i,k}$$

or equivalently, maximizing the log likelihood:

$$\log Lik \propto \log p_{i,k}$$

Or equivalently, minimizing the negative log likelihood loss

$$L_i = -\log p_{i,k}, \quad (y_i = k)$$
The Negative Log-Likelihood Loss

\[ L = \sum_{i=1}^{N} L_i = \sum_{i=1}^{N} \left\{ - \sum_{k=0}^{K-1} r_{i,k} \log p_{i,k} \right\} \]

\[ r_{i,k} = \begin{cases} 1 & \text{if } y_i = k \\ 0 & \text{otherwise} \end{cases} \]
The loss function:

\[ L = \sum_{i=1}^{N} L_i = \sum_{i=1}^{N} \left\{ - \sum_{k=0}^{K-1} r_{i,k} \log p_{i,k} \right\} \]

The first derivative:

\[ \frac{\partial L_i}{\partial F_{i,k}} = -(r_{i,k} - p_{i,k}) \]

The second derivative:

\[ \frac{\partial^2 L_i}{\partial F_{i,k}^2} = p_{i,k} (1 - p_{i,k}) . \]
The Original LogitBoost Algorithm

1: $F_{i,k} = 0, \ p_{i,k} = \frac{1}{K}, \ k = 0 \ to \ K - 1, \ i = 1 \ to \ N$

2: For $m = 1 \ to \ M$ Do

3: For $k = 0 \ to \ K - 1$, Do

4: $w_{i,k} = p_{i,k} \ (1 - p_{i,k}), \ z_{i,k} = \frac{r_{i,k} - p_{i,k}}{p_{i,k} \ (1 - p_{i,k})}$.

5: Fit the function $f_{i,k}$ by a weighted least-square of $z_{i,k}$ to $x_i$ with weights $w_{i,k}$.

6: $F_{i,k} = F_{i,k} + \nu \frac{K - 1}{K} \left( f_{i,k} - \frac{1}{K} \sum_{k=0}^{K-1} f_{i,k} \right)$

7: End

8: $p_{i,k} = \exp(F_{i,k}) / \sum_{s=0}^{K-1} \exp(F_{i,s}), \ k = 0 \ to \ K - 1, \ i = 1 \ to \ N$

9: End
The Original MART Algorithm

1: \( F_{i,k} = 0, \ p_{i,k} = \frac{1}{K}, \ k = 0 \text{ to } K - 1, \ i = 1 \text{ to } N \)

2: For \( m = 1 \) to \( M \) Do

3: For \( k = 0 \) to \( K - 1 \) Do

4: \[ \{ R_{j,k,m} \}_{j=1}^{J} = J \text{-terminal node regression tree from } \{ r_{i,k} - p_{i,k}, \ x_i \}_{i=1}^{N} \]

5: \[ \beta_{j,k,m} = \frac{K-1}{K} \frac{\sum_{x_i \in R_{j,k,m}} r_{i,k} - p_{i,k}}{\sum_{x_i \in R_{j,k,m}} (1-p_{i,k}) p_{i,k}} \]

6: \[ F_{i,k} = F_{i,k} + \nu \sum_{j=1}^{J} \beta_{j,k,m} 1_{x_i \in R_{j,k,m}} \]

7: End

8: \[ p_{i,k} = \exp(F_{i,k}) / \sum_{s=0}^{K-1} \exp(F_{i,s}), \ k = 0 \text{ to } K - 1, \ i = 1 \text{ to } N \]

9: End
Comparing LogitBoost with MART

- LogitBoost used first and second derivatives to construct the trees.

- MART only used the first order information to construct the trees.

- Both used second-order information to update values of the terminal nodes.

- LogitBoost was believed to have numerical instability problems.
The Numerical Issue in LoigtBoost

4: \[ w_{i,k} = p_{i,k} (1 - p_{i,k}), \quad z_{i,k} = \frac{r_{i,k} - p_{i,k}}{p_{i,k} (1 - p_{i,k})}. \]

5: Fit the function \( f_{i,k} \) by a weighted least-square of \( z_{i,k} \) to \( x_i \) with weights \( w_{i,k} \).

6: \[ F_{i,k} = F_{i,k} + \nu \frac{K - 1}{K} \left( f_{i,k} - \frac{1}{K} \sum_{k=0}^{K-1} f_{i,k} \right) \]

The "instability issue":

When \( p_{i,k} \) is close to 0 or 1, \( z_{i,k} = \frac{r_{i,k} - p_{i,k}}{p_{i,k} (1 - p_{i,k})} \) may approach infinity.
(Friedman et al 2000) used regression trees and suggested some “crucial implementation protections”:

- In Line 4, compute $z_{i,k}$ by $\frac{1}{p_{i,k}}$ (if $r_{i,k} = 1$) or $\frac{-1}{1-p_{i,k}}$ (if $r_{i,k} = 0$).

- Bound $|z_{i,k}|$ by $z_{max} \in [2, 4]$.

Robust LogitBoost avoids this pointwise thresholding and is free of numerical problems.

It turns out, the numerical issue essentially does not really exist.
Regression Tree Split Criterion

Square error: $\sum_i (z_i - \bar{z})^2$  
Square error: $\sum_{x_i<s} (z_i - \bar{z}_L)^2 + \sum_{x_i\geq s} (z_i - \bar{z}_R)^2$

We can also use **weighted** square errors.
Tree-Splitting Using the Second-Order Information

**Feature values:** \(x_i, i = 1 \text{ to } N\). Assume \(x_1 \leq x_2 \leq \ldots \leq x_N\).

**Weight values:** \(w_i, i = 1 \text{ to } N\). **Response values:** \(z_i, i = 1 \text{ to } N\).

We seek the index \(s, 1 \leq s < N\), to maximize the gain of weighted SE:

\[
\text{Gain}(s) = SE_T - (SE_L + SE_R)
\]

\[
= \sum_{i=1}^{N} (z_i - \bar{z})^2 w_i - \left[ \sum_{i=1}^{s} (z_i - \bar{z}_L)^2 w_i + \sum_{i=s+1}^{N} (z_i - \bar{z}_R)^2 w_i \right]
\]

where \(\bar{z} = \frac{\sum_{i=1}^{N} z_i w_i}{\sum_{i=1}^{N} w_i}\), \(\bar{z}_L = \frac{\sum_{i=1}^{s} z_i w_i}{\sum_{i=1}^{s} w_i}\), \(\bar{z}_R = \frac{\sum_{i=s+1}^{N} z_i w_i}{\sum_{i=s+1}^{N} w_i}\).
After simplification, we obtain

\[
Gain(s) = \frac{\left[\sum_{i=1}^{s} z_i w_i \right]^2}{\sum_{i=1}^{s} w_i} + \frac{\left[\sum_{i=s+1}^{N} z_i w_i \right]^2}{\sum_{i=s+1}^{N} w_i} - \frac{\left[\sum_{i=1}^{N} z_i w_i \right]^2}{\sum_{i=1}^{N} w_i}
\]

\[
= \frac{\left[\sum_{i=1}^{s} r_{i,k} - p_{i,k} \right]^2}{\sum_{i=1}^{s} p_{i,k}(1 - p_{i,k})} + \frac{\left[\sum_{i=s+1}^{N} r_{i,k} - p_{i,k} \right]^2}{\sum_{i=s+1}^{N} p_{i,k}(1 - p_{i,k})} - \frac{\left[\sum_{i=1}^{N} r_{i,k} - p_{i,k} \right]^2}{\sum_{i=1}^{N} p_{i,k}(1 - p_{i,k})}
\]

Recall \( w_i = p_{i,k}(1 - p_{i,k}) \), \( z_i = \frac{r_{i,k} - p_{i,k}}{p_{i,k}(1 - p_{i,k})} \).

This procedure is numerically stable.
MART only used the first order information to construct the trees:

\[
MARTGain(s) = \frac{1}{s} \left[ \sum_{i=1}^{s} r_{i,k} - p_{i,k} \right]^2 + \frac{1}{N - s} \left[ \sum_{i=s+1}^{N} r_{i,k} - p_{i,k} \right]^2
\]

\[
- \frac{1}{N} \left[ \sum_{i=1}^{N} r_{i,k} - p_{i,k} \right]^2.
\]

Which can also be derived by letting weights \( w_{i,k} = 1 \) and response \( z_{i,k} = r_{i,k} - p_{i,k} \).

LogitBoost used more information and could be more accurate in many datasets.
Robust LogitBoost

1: $F_{i,k} = 0$, $p_{i,k} = \frac{1}{K}$, $k = 0$ to $K - 1$, $i = 1$ to $N$
2: For $m = 1$ to $M$ Do
3: For $k = 0$ to $K - 1$ Do

4: $\{R_{j,k,m}\}_{j=1}^{J} = J$-terminal node regression tree from $\{r_{i,k} - p_{i,k}, \ x_{i}\}_{i=1}^{N}$, with weights $p_{i,k}(1 - p_{i,k})$.

5: $\beta_{j,k,m} = \frac{K-1}{K} \frac{\sum x_{i} \in R_{j,k,m} r_{i,k} - p_{i,k}}{\sum x_{i} \in R_{j,k,m} (1 - p_{i,k}) p_{i,k}}$

6: $F_{i,k} = F_{i,k} + \nu \sum_{j=1}^{J} \beta_{j,k,m} 1 x_{i} \in R_{j,k,m}$
7: End
8: $p_{i,k} = \exp(F_{i,k}) / \sum_{s=0}^{K-1} \exp(F_{i,s})$, $k = 0$ to $K - 1$, $i = 1$ to $N$
9: End
Experiments on Binary Classification

(Multi-class classification is even more interesting!)

Data

**IJCNN1**: 49990 training samples, 91701 test samples

This dataset was used in a competition. LibSVM was the winner.

----------

**Forest100k**: 100000 training samples, 50000 test samples

**Forest521k**: 521012 training samples, 50000 test samples

The two largest datasets from Bordes et al. JMLR 2005,
Fast Kernel Classifiers with Online and Active Learning
Test misclassification error as a function of iterations for different algorithms:

- **MART**: Red line
- **LibSVM**: Blue dashed line
- **Robust LogitBoost**: Black line

**Test: J = 20 \( \nu = 0.1 \)**
Forest100k Test Errors

Test: $J = 20 \; \nu = 0.1$

Test misclassification error

Iterations

Test misclassification error

Robust LogitBoost

SVM

MART
Test: $J = 20$ $\nu = 0.1$

Forest521k Test Errors

Test misclassification error

Robust LogitBoost

SVM

MART

Iterations

Test misclassification error

0 5000 10000

1500 2000 2500 3000 3500 4000
ABC-Boost for Multi-Class Classification

**ABC** = Adaptive Base Class

**ABC-MART** = ABC-Boost + MART

**ABC-LogitBoost** = ABC-Boost + (Robust) LogitBoost

The key to the success of ABC-Boost is the use of “better” derivatives.
Review Components of Logistic Regression

The multinomial logit probability model:

\[ p_k = \frac{e^{F_k}}{\sum_{s=0}^{K-1} e^{F_s}}, \quad \sum_{k=0}^{K-1} p_k = 1 \]

where \( F_k = F_k(x) \) is the function to be learned from the data.

The sum-to-zero constraint:

\[ \sum_{k=0}^{K-1} F_k(x) = 0 \]

is commonly used to obtain a unique solution (only \( K - 1 \) degrees of freedom).
Why the sum-to-zero constraint?

\[
\frac{e^{F_{i,k}} + C}{\sum_{s=0}^{K-1} e^{F_{i,s}} + C} = \frac{e^C e^{F_{i,k}}}{e^C \sum_{s=0}^{K-1} e^{F_{i,s}}} = \frac{e^{F_{i,k}}}{\sum_{s=0}^{K-1} e^{F_{i,s}}} = p_{i,k}.
\]

For identifiability, one should impose a constraint.

One popular choice is to assume \( \sum_{k=0}^{K-1} F_{i,k} = \text{const} \), equivalent to

\[
\sum_{k=0}^{K-1} F_{i,k} = 0.
\]

This is the assumption used in many papers including LogitBoost and MART.
The negative log-Likelihood loss

\[ L = \sum_{i=1}^{N} L_i = \sum_{i=1}^{N} \left\{ - \sum_{k=0}^{K-1} r_{i,k} \log p_{i,k} \right\} \]

\[ r_{i,k} = \begin{cases} 1 & \text{if } y_i = k \\ 0 & \text{otherwise} \end{cases} \]

\[ \sum_{k=0}^{K-1} r_{i,k} = 1 \]
Derivatives used in LogitBoost and MART:

\[
\frac{\partial L_i}{\partial F_{i,k}} = -(r_{i,k} - p_{i,k})
\]

\[
\frac{\partial^2 L_i}{\partial F_{i,k}^2} = p_{i,k} (1 - p_{i,k})
\]

These are taught in statistics textbooks.
Derivatives Under Sum-to-zero Constraint

The loss function:

\[ L_i = - \sum_{k=0}^{K-1} r_{i,k} \log p_{i,k} \]

The probability model and sum-to-zero constraint:

\[ p_{i,k} = \frac{e^{F_{i,k}}}{\sum_{s=0}^{K-1} e^{F_{i,s}}}, \quad \sum_{k=0}^{K-1} F_{i,k} = 0 \]

Without loss of generality, we assume \( k = 0 \) is the base class

\[ F_{i,0} = - \sum_{k=1}^{K-1} F_{i,k} \]
New derivatives:

\[
\frac{\partial L_i}{\partial F_{i,k}} = (r_{i,0} - p_{i,0}) - (r_{i,k} - p_{i,k}),
\]

\[
\frac{\partial^2 L_i}{\partial F^2_{i,k}} = p_{i,0}(1 - p_{i,0}) + p_{i,k}(1 - p_{i,k}) + 2p_{i,0}p_{i,k}.
\]

MART and LogitBoost used:

\[
\frac{\partial L_i}{\partial F_{i,k}} = -(r_{i,k} - p_{i,k}), \quad \frac{\partial^2 L_i}{\partial F^2_{i,k}} = p_{i,k}(1 - p_{i,k}).
\]
Adaptive Base Class Boost (ABC-Boost)

Two Key Ideas of ABC-Boost:

1. Formulate the multi-class boosting algorithm by considering a base class.

   Do not have to train for the base class, which is inferred from the sum-zero-constraint
   \[
   \sum_{k=0}^{K-1} F_{i,k} = 0.
   \]

2. At each boosting step, adaptively choose the base class.
How to choose the base class?

- **(One Idea)** Exhaustively search for all $K$ base classes and choose the base class that leads to the best performance (smallest training loss). Computationally more expensive

- **(Another Idea)** At each iteration, choose the base class which has the worst performance (largest training loss). Computationally efficient (occasionally slightly worse performance)

- Other ideas.
\[ \text{ABC-MART} = \text{ABC-Boost} + \text{MART}. \]

\[ \text{ABC-LogitBoost} = \text{ABC-Boost} + \text{Robust LogitBoost}. \]
The Original MART Algorithm

1: $F_{i,k} = 0, \ p_{i,k} = \frac{1}{K}, \ k = 0 \text{ to } K - 1, \ i = 1 \text{ to } N$

2: For $m = 1 \text{ to } M$ Do

3: \ For $k = 0 \text{ to } K - 1$ Do

4: \ \{R_{j,k,m}\}_{j=1}^{J} = J\text{-terminal node regression tree from } \{r_{i,k} - p_{i,k}, \ x_i\}_{i=1}^{N}$

5: $\beta_{j,k,m} = \frac{K-1}{K} \frac{\sum_{x_i \in R_{j,k,m}} r_{i,k} - p_{i,k}}{\sum_{x_i \in R_{j,k,m}} (1-p_{i,k}) p_{i,k}}$

6: $F_{i,k} = F_{i,k} + \nu \sum_{j=1}^{J} \beta_{j,k,m} 1_{x_i \in R_{j,k,m}}$

7: \ End

8: $p_{i,k} = \exp(F_{i,k}) / \sum_{s=0}^{K-1} \exp(F_{i,s}), \ k = 0 \text{ to } K - 1, \ i = 1 \text{ to } N$

9: End
1: $F_{i,k} = 0, k = 0$ to $K - 1, i = 1$ to $N$

2: For $m = 1$ to $M$ Do

: For $b = 0$ to $K - 1$ DO

3: For $k = 0$ to $K - 1$ (and $k \neq b$) Do

4: \[ \{R_{j,k,m}\}_{j=1}^{J} = J\text{-terminal node tree from } \{r_{i,k} - p_{i,k} - (r_{i,b} - p_{i,b}), \ x_i\}_{i=1}^{N} \]

5: \[ \beta_{j,k,m} = \frac{\sum_{x_i \in R_{j,k,m}} (r_{i,k} - p_{i,k}) - (r_{i,b} - p_{i,b})}{\sum_{x_i \in R_{j,k,m}} (1 - p_{i,k}) p_{i,k} + (1 - p_{i,b}) p_{i,b} + 2 p_{i,k} p_{i,b}} \]

6: \[ F_{i,k} = F_{i,k} + \nu \sum_{j=1}^{J} \beta_{j,k,m} 1_{x_i \in R_{j,k,m}} \]

7: End

8: \[ p_{i,k} = \exp(F_{i,k}) / \sum_{s=0}^{K-1} \exp(F_{i,s}) \]

9: End
Datasets

- **UCI-Covertype**  Total 581012 samples.
  Two datasets were generated: *Covertype290k, Covertype145k*

- **UCI-Poker**  Original 25010 training samples and 1 million test samples.
  *Poker25kT1, Poker25kT2, Poker525k, Poker275k, Poker150k, Poker100k*.

- **MNIST**  Originally 60000 training samples and 10000 test samples.
  *MNIST10k* swapped the training with test samples.

- **Many variations of MNIST**  Original MNIST is a well-known easy problem. ([www.iro.umontreal.ca/~lisa/twiki/bin/view.cgi/Public/DeepVsShallowComparisonICML2007](http://www.iro.umontreal.ca/~lisa/twiki/bin/view.cgi/Public/DeepVsShallowComparisonICML2007)) created a variety of much more difficult datasets by adding various background (correlated) noise, background images, rotations, etc.

- **UCI-Letter**  Total 20000 samples.
<table>
<thead>
<tr>
<th>dataset</th>
<th>$K$</th>
<th># training</th>
<th># test</th>
<th># features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Covertype290k</td>
<td>7</td>
<td>290506</td>
<td>290506</td>
<td>54</td>
</tr>
<tr>
<td>Covertype145k</td>
<td>7</td>
<td>145253</td>
<td>290506</td>
<td>54</td>
</tr>
<tr>
<td>Poker525k</td>
<td>10</td>
<td>525010</td>
<td>500000</td>
<td>25</td>
</tr>
<tr>
<td>Poker275k</td>
<td>10</td>
<td>275010</td>
<td>500000</td>
<td>25</td>
</tr>
<tr>
<td>Poker150k</td>
<td>10</td>
<td>150010</td>
<td>500000</td>
<td>25</td>
</tr>
<tr>
<td>Poker100k</td>
<td>10</td>
<td>100010</td>
<td>500000</td>
<td>25</td>
</tr>
<tr>
<td>Poker25kT1</td>
<td>10</td>
<td>25010</td>
<td>500000</td>
<td>25</td>
</tr>
<tr>
<td>Poker25kT2</td>
<td>10</td>
<td>25010</td>
<td>500000</td>
<td>25</td>
</tr>
<tr>
<td>Mnist10k</td>
<td>10</td>
<td>10000</td>
<td>60000</td>
<td>784</td>
</tr>
<tr>
<td>M-Basic</td>
<td>10</td>
<td>12000</td>
<td>50000</td>
<td>784</td>
</tr>
<tr>
<td>M-Rotate</td>
<td>10</td>
<td>12000</td>
<td>50000</td>
<td>784</td>
</tr>
<tr>
<td>M-Image</td>
<td>10</td>
<td>12000</td>
<td>50000</td>
<td>784</td>
</tr>
<tr>
<td>M-Rand</td>
<td>10</td>
<td>12000</td>
<td>50000</td>
<td>784</td>
</tr>
<tr>
<td>M-RotImg</td>
<td>10</td>
<td>12000</td>
<td>50000</td>
<td>784</td>
</tr>
<tr>
<td>M-Noise1</td>
<td>10</td>
<td>10000</td>
<td>2000</td>
<td>784</td>
</tr>
<tr>
<td>M-Noise2</td>
<td>10</td>
<td>10000</td>
<td>2000</td>
<td>784</td>
</tr>
<tr>
<td>M-Noise3</td>
<td>10</td>
<td>10000</td>
<td>2000</td>
<td>784</td>
</tr>
<tr>
<td>M-Noise4</td>
<td>10</td>
<td>10000</td>
<td>2000</td>
<td>784</td>
</tr>
<tr>
<td>M-Noise5</td>
<td>10</td>
<td>10000</td>
<td>2000</td>
<td>784</td>
</tr>
<tr>
<td>M-Noise6</td>
<td>10</td>
<td>10000</td>
<td>2000</td>
<td>784</td>
</tr>
<tr>
<td>Letter15k</td>
<td>26</td>
<td>15000</td>
<td>5000</td>
<td>16</td>
</tr>
<tr>
<td>Letter4k</td>
<td>26</td>
<td>4000</td>
<td>16000</td>
<td>16</td>
</tr>
<tr>
<td>Letter2k</td>
<td>26</td>
<td>2000</td>
<td>18000</td>
<td>16</td>
</tr>
</tbody>
</table>
## Summary of test mis-classification errors

<table>
<thead>
<tr>
<th>Dataset</th>
<th>mart</th>
<th>abc-mart</th>
<th>logitboost</th>
<th>abc-logitboost</th>
<th>logistic regression</th>
<th># test</th>
</tr>
</thead>
<tbody>
<tr>
<td>Covertype290k</td>
<td>11350</td>
<td>10454</td>
<td>10765</td>
<td>9727</td>
<td>80233</td>
<td>290506</td>
</tr>
<tr>
<td>Covertype145k</td>
<td>15767</td>
<td>14665</td>
<td>14928</td>
<td>13986</td>
<td>80314</td>
<td>290506</td>
</tr>
<tr>
<td>Poker525k</td>
<td>7061</td>
<td>2424</td>
<td>2704</td>
<td>1736</td>
<td>248892</td>
<td>500000</td>
</tr>
<tr>
<td>Poker275k</td>
<td>15404</td>
<td>3679</td>
<td>6533</td>
<td>2727</td>
<td>248892</td>
<td>500000</td>
</tr>
<tr>
<td>Poker150k</td>
<td>22289</td>
<td>12340</td>
<td>16163</td>
<td>5104</td>
<td>248892</td>
<td>500000</td>
</tr>
<tr>
<td>Poker100k</td>
<td>27871</td>
<td>21293</td>
<td>25715</td>
<td>13707</td>
<td>248892</td>
<td>500000</td>
</tr>
<tr>
<td>Poker25kT1</td>
<td>43575</td>
<td>34879</td>
<td>46789</td>
<td>37345</td>
<td>250110</td>
<td>500000</td>
</tr>
<tr>
<td>Poker25kT2</td>
<td>42935</td>
<td>34326</td>
<td>46600</td>
<td>36731</td>
<td>249056</td>
<td>500000</td>
</tr>
<tr>
<td>Mnist10k</td>
<td>2815</td>
<td>2440</td>
<td>2381</td>
<td>2102</td>
<td>13950</td>
<td>600000</td>
</tr>
<tr>
<td>M-Basic</td>
<td>2058</td>
<td>1843</td>
<td>1723</td>
<td>1602</td>
<td>10993</td>
<td>500000</td>
</tr>
<tr>
<td>M-Rotate</td>
<td>7674</td>
<td>6634</td>
<td>6813</td>
<td>5959</td>
<td>26584</td>
<td>500000</td>
</tr>
<tr>
<td>M-Image</td>
<td>5821</td>
<td>4727</td>
<td>4703</td>
<td>4268</td>
<td>19353</td>
<td>500000</td>
</tr>
<tr>
<td>M-Rand</td>
<td>6577</td>
<td>5300</td>
<td>5020</td>
<td>4725</td>
<td>18189</td>
<td>500000</td>
</tr>
<tr>
<td>M-RotImg</td>
<td>24912</td>
<td>23072</td>
<td>22962</td>
<td>22343</td>
<td>33216</td>
<td>500000</td>
</tr>
<tr>
<td>M-Noise1</td>
<td>305</td>
<td>245</td>
<td>267</td>
<td>234</td>
<td>935</td>
<td>2000</td>
</tr>
<tr>
<td>M-Noise2</td>
<td>325</td>
<td>262</td>
<td>270</td>
<td>237</td>
<td>940</td>
<td>2000</td>
</tr>
<tr>
<td>M-Noise3</td>
<td>310</td>
<td>264</td>
<td>277</td>
<td>238</td>
<td>954</td>
<td>2000</td>
</tr>
<tr>
<td>M-Noise4</td>
<td>308</td>
<td>243</td>
<td>256</td>
<td>238</td>
<td>933</td>
<td>2000</td>
</tr>
<tr>
<td>M-Noise5</td>
<td>294</td>
<td>244</td>
<td>242</td>
<td>227</td>
<td>867</td>
<td>2000</td>
</tr>
<tr>
<td>M-Noise6</td>
<td>279</td>
<td>224</td>
<td>226</td>
<td>201</td>
<td>788</td>
<td>2000</td>
</tr>
<tr>
<td>Letter15k</td>
<td>155</td>
<td>125</td>
<td>139</td>
<td>109</td>
<td>1130</td>
<td>50000</td>
</tr>
<tr>
<td>Letter4k</td>
<td>1370</td>
<td>1149</td>
<td>1252</td>
<td>1055</td>
<td>3712</td>
<td>16000</td>
</tr>
<tr>
<td>Letter2k</td>
<td>2482</td>
<td>2220</td>
<td>2309</td>
<td>2034</td>
<td>4381</td>
<td>18000</td>
</tr>
</tbody>
</table>
Comparisons with SVM and Deep Learning

Datasets: M-Noise1 to M-Noise6

Results on SVM, Neural Nets, and Deep Learning are from

www.iro.umontreal.ca/~lisa/twiki/bin/view.cgi/Public/DeepVsShallowComparisonICML2007
Comparisons with SVM and Deep Learning

Datasets: M-Noise1 to M-Noise6
<table>
<thead>
<tr>
<th>Model</th>
<th>M-Basic</th>
<th>M-Rotate</th>
<th>M-Image</th>
<th>M-Rand</th>
<th>M-RotImg</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVM-RBF</td>
<td>3.05%</td>
<td>11.11%</td>
<td>22.61%</td>
<td>14.58%</td>
<td>55.18%</td>
</tr>
<tr>
<td>SVM-POLY</td>
<td>3.69%</td>
<td>15.42%</td>
<td>24.01%</td>
<td>16.62%</td>
<td>56.41%</td>
</tr>
<tr>
<td>NNET</td>
<td>4.69%</td>
<td>18.11%</td>
<td>27.41%</td>
<td>20.04%</td>
<td>62.16%</td>
</tr>
<tr>
<td>DBN-3</td>
<td>3.11%</td>
<td>10.30%</td>
<td>16.31%</td>
<td>6.73%</td>
<td>47.39%</td>
</tr>
<tr>
<td>SAA-3</td>
<td>3.46%</td>
<td>10.30%</td>
<td>23.00%</td>
<td>11.28%</td>
<td>51.93%</td>
</tr>
<tr>
<td>DBN-1</td>
<td>3.94%</td>
<td>14.69%</td>
<td>16.15%</td>
<td>9.80%</td>
<td>52.21%</td>
</tr>
<tr>
<td>mart</td>
<td>4.12%</td>
<td>15.35%</td>
<td>11.64%</td>
<td>13.15%</td>
<td>49.82%</td>
</tr>
<tr>
<td>abc-mart</td>
<td>3.69%</td>
<td>13.27%</td>
<td>9.45%</td>
<td>10.60%</td>
<td>46.14%</td>
</tr>
<tr>
<td>logitboost</td>
<td>3.45%</td>
<td>13.63%</td>
<td>9.41%</td>
<td>10.04%</td>
<td>45.92%</td>
</tr>
<tr>
<td>abc-logitboost</td>
<td>3.20%</td>
<td>11.92%</td>
<td>8.54%</td>
<td>9.45%</td>
<td>44.69%</td>
</tr>
</tbody>
</table>
Training Loss Vs Boosting Iterations

Train

Poker525k: J=20, ν=0.1

abc−logit

abc−mart

logit

mart
Test Errors Vs Boosting Iterations

Mnist10k: $J = 20$, $\nu = 0.1$

Letter15k: $J = 20$, $\nu = 0.1$
Letter4k: $J = 20, \nu = 0.1$

Letter2k: $J = 20, \nu = 0.1$
Detailed Experiment Results on Mnist10k

\[ J \in \{4, 6, 8, 10, 12, 14, 16, 18, 20, 24, 30, 40, 50\} \]
\[ \nu \in \{0.04, 0.06, 0.08, 0.1\} \].

The goal is to show the improvements at all reasonable combinations of \( J \) and \( \nu \).
<table>
<thead>
<tr>
<th></th>
<th>mart</th>
<th>abc-mart</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\nu = 0.04$</td>
<td>3356 <strong>3060</strong></td>
<td>3329 <strong>3019</strong></td>
</tr>
<tr>
<td>$J = 4$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\nu = 0.06$</td>
<td>3185 <strong>2760</strong></td>
<td>3093 <strong>2626</strong></td>
</tr>
<tr>
<td>$J = 6$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\nu = 0.08$</td>
<td>3049 <strong>2558</strong></td>
<td>3054 <strong>2555</strong></td>
</tr>
<tr>
<td>$J = 8$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\nu = 0.1$</td>
<td>3020 <strong>2547</strong></td>
<td>2973 <strong>2521</strong></td>
</tr>
<tr>
<td>$J = 10$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\nu = 0.04$</td>
<td>2927 <strong>2498</strong></td>
<td>2917 <strong>2457</strong></td>
</tr>
<tr>
<td>$J = 12$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\nu = 0.06$</td>
<td>2925 <strong>2487</strong></td>
<td>2901 <strong>2471</strong></td>
</tr>
<tr>
<td>$J = 14$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\nu = 0.08$</td>
<td>2899 <strong>2478</strong></td>
<td>2893 <strong>2452</strong></td>
</tr>
<tr>
<td>$J = 16$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\nu = 0.1$</td>
<td>2857 <strong>2469</strong></td>
<td>2880 <strong>2460</strong></td>
</tr>
<tr>
<td>$J = 18$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\nu = 0.04$</td>
<td>2833 <strong>2441</strong></td>
<td>2834 <strong>2448</strong></td>
</tr>
<tr>
<td>$J = 20$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\nu = 0.06$</td>
<td>2840 <strong>2447</strong></td>
<td>2827 <strong>2431</strong></td>
</tr>
<tr>
<td>$J = 24$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\nu = 0.08$</td>
<td>2826 <strong>2457</strong></td>
<td>2822 <strong>2443</strong></td>
</tr>
<tr>
<td>$J = 30$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\nu = 0.1$</td>
<td>2837 <strong>2482</strong></td>
<td>2809 <strong>2440</strong></td>
</tr>
<tr>
<td>$J = 40$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\nu = 0.04$</td>
<td>2813 <strong>2502</strong></td>
<td>2826 <strong>2459</strong></td>
</tr>
<tr>
<td>$J = 50$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>logitboost</td>
<td>abc-logit</td>
</tr>
<tr>
<td>-------</td>
<td>------------</td>
<td>-----------</td>
</tr>
<tr>
<td></td>
<td>$\nu = 0.04$</td>
<td>$\nu = 0.06$</td>
</tr>
<tr>
<td>$J = 4$</td>
<td>2936 2630</td>
<td>2970 2600</td>
</tr>
<tr>
<td>$J = 6$</td>
<td>2710 2263</td>
<td>2693 2252</td>
</tr>
<tr>
<td>$J = 8$</td>
<td>2599 2159</td>
<td>2619 2138</td>
</tr>
<tr>
<td>$J = 10$</td>
<td>2553 2122</td>
<td>2527 2118</td>
</tr>
<tr>
<td>$J = 12$</td>
<td>2472 2084</td>
<td>2468 2090</td>
</tr>
<tr>
<td>$J = 14$</td>
<td>2451 2083</td>
<td>2420 2094</td>
</tr>
<tr>
<td>$J = 16$</td>
<td>2424 2111</td>
<td>2437 2114</td>
</tr>
<tr>
<td>$J = 18$</td>
<td>2399 2088</td>
<td>2402 2087</td>
</tr>
<tr>
<td>$J = 20$</td>
<td>2388 2128</td>
<td>2414 2112</td>
</tr>
<tr>
<td>$J = 24$</td>
<td>2442 2174</td>
<td>2415 2147</td>
</tr>
<tr>
<td>$J = 30$</td>
<td>2468 2235</td>
<td>2434 2237</td>
</tr>
<tr>
<td>$J = 40$</td>
<td>2551 2310</td>
<td>2509 2284</td>
</tr>
<tr>
<td>$J = 50$</td>
<td>2612 2353</td>
<td>2622 2359</td>
</tr>
</tbody>
</table>
Mnist10k: \( J = 10, \nu = 0.04 \)

Mnist10k: \( J = 10, \nu = 0.06 \)

Mnist10k: \( J = 10, \nu = 0.1 \)

Mnist10k: \( J = 12, \nu = 0.04 \)

Mnist10k: \( J = 12, \nu = 0.06 \)

Mnist10k: \( J = 12, \nu = 0.1 \)

Mnist10k: \( J = 14, \nu = 0.04 \)

Mnist10k: \( J = 14, \nu = 0.06 \)

Mnist10k: \( J = 14, \nu = 0.1 \)
Want to See More?
Drawbacks of Trees & Boosting Algorithms

- Not suitable for nominal categorical variables with many (e.g., billion) categories
- Training is slow and parallel implementations are energy-consuming
- Model size is typically large (i.e., many trees) if we need high accuracy
- In some cases, trees can not reach the accuracy of DNN (deep learning), although the results of trees are almost never bad.
Comparisons with SVM and Deep Learning

Datasets: M-Noise1 to M-Noise6
## More Comparisons with SVM and Deep Learning

<table>
<thead>
<tr>
<th>Model</th>
<th>M-Basic</th>
<th>M-Rotate</th>
<th>M-Image</th>
<th>M-Rand</th>
<th>M-RotImg</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVM-RBF</td>
<td>3.05%</td>
<td>11.11%</td>
<td>22.61%</td>
<td>14.58%</td>
<td>55.18%</td>
</tr>
<tr>
<td>SVM-POLY</td>
<td>3.69%</td>
<td>15.42%</td>
<td>24.01%</td>
<td>16.62%</td>
<td>56.41%</td>
</tr>
<tr>
<td>NNET</td>
<td>4.69%</td>
<td>18.11%</td>
<td>27.41%</td>
<td>20.04%</td>
<td>62.16%</td>
</tr>
<tr>
<td>DBN-3</td>
<td>3.11%</td>
<td><strong>10.30%</strong></td>
<td>16.31%</td>
<td><strong>6.73%</strong></td>
<td>47.39%</td>
</tr>
<tr>
<td>SAA-3</td>
<td>3.46%</td>
<td><strong>10.30%</strong></td>
<td>23.00%</td>
<td>11.28%</td>
<td>51.93%</td>
</tr>
<tr>
<td>DBN-1</td>
<td>3.94%</td>
<td>14.69%</td>
<td>16.15%</td>
<td>9.80%</td>
<td>52.21%</td>
</tr>
<tr>
<td>mart</td>
<td>4.12%</td>
<td>15.35%</td>
<td>11.64%</td>
<td>13.15%</td>
<td>49.82%</td>
</tr>
<tr>
<td>abc-mart</td>
<td>3.69%</td>
<td>13.27%</td>
<td>9.45%</td>
<td>10.60%</td>
<td>46.14%</td>
</tr>
<tr>
<td>logitboost</td>
<td>3.45%</td>
<td>13.63%</td>
<td>9.41%</td>
<td>10.04%</td>
<td>45.92%</td>
</tr>
<tr>
<td>abc-logitboost</td>
<td>3.20%</td>
<td>11.92%</td>
<td><strong>8.54%</strong></td>
<td>9.45%</td>
<td><strong>44.69%</strong></td>
</tr>
</tbody>
</table>
Every (one-dim) feature becomes a vector of naturally ordered integers. This largely simplifies the implementation of trees and also significantly improves the train speed.
Adaptive binning (which is one strategy for building histogram) speeds up training without affecting the performance.

Ref: P. Li, et. al., McRank: Learning to Rank with Multiple Classification and Gradient Boosting, NIPS 2007.
Most implementations of trees used “balanced trees” by controlling the “height” of tree. However, the “best-first” strategy ought to be a better implementation (which was used in the 2007 NIPS McRank paper and later papers on trees).

“Gain” (eg $g = 100$) is the potential achievable gain if we choose to split on that region. In this example, the region with ($g = 40$) will be the next one to split on.
**Advantage of Best-First Split Criterion**

- Implementation is simple. A tree with $J$ terminal nodes needs a vector of fixed length of $2J - 1$ entries. The parent-children relationship is also convenient.

- It is a natural process. Split data where a split is most needed.

- No need to restrict oneself to 8, 16, 32, ..., leaves any more. This is a tremendous convenience in practice when choosing tuning parameters.

- Using a balanced tree can potentially cause “data imbalance” problem.

The 2007 NIPS McRank paper and all “abc-boost” papers used this implementation.
Hashing algorithms for machine learning

Hashing + logistic regression and hashing + DNN can often be nice combinations:

- Hashing for dealing with ultra-high-dimensional data
- Hashing for building compact (e.g., single machine) learning models
- Hashing for building more complex (and more accurate) learning models
- Hashing as a feature engineering tool
- Hashing for building nonlinear learning models at linear cost
One Major Source of High-Dimensional Data: **Histogram**

Histogram-based features are very popular in practice, for example, natural language processing (NLP) and computer vision.

It can be viewed as high-dimensional vector: $u_i \geq 0, \ i = 1, 2, \ldots, D$

The size of the space $D$ can often be extremely large. For example, $D$ can be the total number of words, or combinations of words (or characters, or visual words).

In search industry, $D = 2^{64}$ is often used, for convenience.
An Example of Text Data Representation by $n$-grams

Each document (Web page) can be viewed as a set of $n$-grams.

For example, after parsing, a sentence “today is a nice day” becomes

- $n = 1$: \{“today”, “is”, “a”, “nice”, “day”\}
- $n = 2$: \{“today is”, “is a”, “a nice”, “nice day”\}
- $n = 3$: \{“today is a”, “is a nice”, “a nice day”\}

It is common to use $n \geq 5$.

Using $n$-grams generates extremely high dimensional vectors, e.g., $D = (10^5)^n$. \((10^5)^5 = 10^{25} = 2^{83}\), although in current practice, it seems $D = 2^{64}$ suffices.

As a highly successful practice, $n$-gram representations have many variants, e.g., word $n$-grams, character $n$-grams, skip $n$-grams, etc.
**Webspam: A Small Example of \( n \)-gram Data**

**Task:** Classifying 350K documents into spam or non-spam (binary classification).

<table>
<thead>
<tr>
<th>Model Configuration</th>
<th>Dim. (D)</th>
<th>Training Time</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-gram + linear SVM</td>
<td>254</td>
<td>20 sec</td>
<td>93.30%</td>
</tr>
<tr>
<td>3-gram + linear SVM</td>
<td>16,609,143</td>
<td>200 sec</td>
<td>99.6%</td>
</tr>
<tr>
<td>3-gram + kernel SVM</td>
<td>16,609,143</td>
<td>About a Week</td>
<td>99.6%</td>
</tr>
</tbody>
</table>

---

**(Character) 1-gram:** Frequencies of occurrences of single characters.

**(Character) 3-gram:** Frequencies of occurrences of 3-contiguous characters.
Challenges with Using High-Dimensional Features

- **High dimensionality** This may lead to large & costly (in both training and testing) statistical models and create large search space.

- **High storage cost** If the (expanded) features are fully materialized, they might be way too large to store/transmit, even for very sparse data.

- **Streaming data** Histogram is a streaming model. How to compute summaries without storing/materializing the entire histograms, and how to update summary statistics without accessing the original histograms?

- **Binary vs. non-binary** While in NLP and search it is popular to use very high-dimensional and binary representations, the current mainstream practice in computer vision is to use non-binary features. In general, binary representations require a much large space (dimensionality).
Similarity of Histograms and Chi-Square Kernel

Given two high-dim nonnegative vectors \( u, v \in \mathbb{R}^D \), the chi-square similarity is

\[
\rho_{\chi^2} = \sum_{i=1}^{D} \frac{2u_i v_i}{u_i + v_i},
\]

\[
\sum_{i=1}^{D} u_i = \sum_{i=1}^{D} v_i = 1
\]

The chi-square similarity is closely related to the chi-square distance \( d_{\chi^2} \):

\[
d_{\chi^2} = \sum_{i=1}^{D} \frac{(u_i - v_i)^2}{u_i + v_i} = \sum_{i=1}^{D} (u_i + v_i) - \sum_{i=1}^{D} \frac{4u_i v_i}{u_i + v_i} = 2 - 2\rho_{\chi^2}
\]

It is a “symmetric” version of the usual chi-square statistic.
Chi-square Kernels

1. $\chi^2$-kernel: $K(u, v) = \rho \chi^2 = \sum_{i=1}^{D} \frac{2u_i v_i}{u_i + v_i}$

2. acos-$\chi^2$-kernel: $K(u, v) = 1 - \frac{1}{\pi} \cos^{-1} \rho \chi^2$

Both kernels are positive definite.
Advantage of Chi-Square Kernels: An Example

**MNIST-Small:** Chi-square kernels substantially improve linear kernel

\[ l_2 \text{-regularized kernel SVM with a regularization parameter } C. \]

MNIST-small: original testing data and merely 1/6 of original training data
Review Logistic Regression and Linear SVM

For example, consider dataset \( \{(x_i, y_i)\}_{i=1}^{n}, x_i \in \mathbb{R}^D, y_i \in \{-1, 1\} \).

One can fit an \(L_2\)-regularized linear logistic regression:

\[
\min_{w} \quad \frac{1}{2} w^T w + C \sum_{i=1}^{n} \log \left( 1 + e^{-y_i w^T x_i} \right),
\]

or the \(L_2\)-regularized linear SVM:

\[
\min_{w} \quad \frac{1}{2} w^T w + C \sum_{i=1}^{n} \max \left\{ 1 - y_i w^T x_i, 0 \right\},
\]

where \(C > 0\) is the penalty (regularization) parameter.
Challenges with Nonlinear Kernel Learning

Kernels were believed not (directly) useful for large-scale applications:

1. Computing kernels is very expensive.

2. Computing a full kernel matrix is wasteful, because not all pairwise kernel values are used during training.

3. The kernel matrix does not fit in memory. The cost of storing the full kernel matrix in the memory is $O(n^2)$, which is not realistic for most PCs even for merely $10^5$. Thus, kernel evaluations are often conducted on the fly, which means the computational cost is dominated by kernel evaluations.

4. In fact, evaluating kernels on-demand would encounter another serious (and often common) issue if the dataset itself is too big for the memory.
Sign Cauchy Random Projections

**A ∈ \(\mathbb{R}^{n \times D}\):** original data matrix, (e.g.,) generated from histograms.

\[
\begin{pmatrix}
A \\
\times \\
R \\
\end{pmatrix} = B
\]

**R ∈ \(\mathbb{R}^{D \times k}\):** random matrix with entries sampled from a standard Cauchy. **B ∈ \(\mathbb{R}^{n \times k}\):** resultant projected matrix, which is much smaller than **A**.

**Sign Cauchy Projections:** only store the signs of the projected data in **B**.

\(u, v ∈ \mathbb{R}^D\): first two rows in **A**. \(x, y ∈ \mathbb{R}^k\): first two rows in **B**.

\[
x = u \times R, \quad y = v \times R.
\]
The Collision Probability

\[ x = u \times \mathbb{R}, \quad y = v \times \mathbb{R}. \]

The Collision Probability is related to the \( \chi^2 \) similarity between \( u \) and \( v \):

\[ \Pr(\text{sign}(x) \neq \text{sign}(y)) \approx \frac{1}{\pi} \cos^{-1}(\rho_{\chi^2}(u, v)) \]

which might be a surprising finding.

In general, the collision probability should be a monotone function of the similarity.
Applications of Sign Cauchy Projections

- **Efficient linear learning (e.g., linear SVM) for $\chi^2$ kernel.**
  
  A negative sign can be coded as “01” and a positive sign as “10” (i.e., a vector of length 2). Concatenate $k$ short vectors to form a vector of length $2^k$.

- **Sub-linear time near-neighbor search in $\chi^2$ similarity.**

  We can code a negative sign by “0” and positive sign by “1” and concatenate $k$ such bits to form a hash table of $2^k$ buckets. In the query phase, one only searches for similar vectors in one bucket.

- **Other applications** requiring computing $\chi^2$ similarity fast using small space.
Sign Cauchy Projections for Statistical Learning

Original data vectors: \( u \in \mathbb{R}^D, \ v \in \mathbb{R}^D \)

Cauchy projection matrix: \( R \in \mathbb{R}^{D \times k} \) \((k = 4\) in this example\)

Sign Cauchy random projections & expansions:

\[
x = u \times R : \quad -61.83 \quad 2.45 \quad 13.83 \quad -1322.05
\]

\[\text{sgn}(x) : \quad -1 \quad +1 \quad +1 \quad -1\]

Expansion : \(01 \quad 10 \quad 10 \quad 01\)

\[
y = v \times R : \quad -11.64 \quad 936.91 \quad -343.43 \quad -12.45
\]

\[\text{sgn}(y) : \quad -1 \quad +1 \quad -1 \quad -1\]

Expansion : \(01 \quad 10 \quad 01 \quad 01\)

Output vectors: binary vectors in \(2k = 8\) dimensions.
Why Expansion?

\[
\text{sgn}(x) : \begin{array}{cccc}
-1 & +1 & +1 & -1 \\
\end{array}
\]

Expansion : 01 10 10 01

\[
\text{sgn}(y) : \begin{array}{cccc}
-1 & +1 & -1 & -1 \\
\end{array}
\]

Expansion : 01 10 01 01

**Inner product** between \([0 \ 1 \ 1 \ 0 \ 1 \ 0 \ 0 \ 1]\) and \([0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 0 \ 1]\) is 3

\[
\frac{3}{k} \text{ is exactly } \frac{1}{k} \sum_{j=1}^{k=4} 1 \{\text{sgn}(x_j) = \text{sgn}(y_j)\} = \frac{3}{4}, \text{ which is the empirical estimate of the collision probability.}
\]

**Extremely efficient batch or online linear algorithms available for inner product (linear kernel) space**
Linear SVM for Approximating Nonlinear Kernel

“negative sign” $\mapsto$ “01”,  “positive sign” $\mapsto$ “10”.

$k$ Cauchy projections $\mapsto$ a binary vector of length $2k$

The inner product approximates the $\text{acos-} \chi^2$-kernel: $1 - \frac{1}{\pi} \cos^{-1} \rho \chi^2$.

Red dashed curve: Classification accuracy of “acos-$\chi^2$-kernel” using LIBSVM.
Solid curves: Classification accuracy of linear SVM with $k$ sign Cauchy projections
Hashing for Machine Learning: What is Happening?

Given two (high-dim) data vectors $u$ and $v$. A carefully designed (random) hash function $h$, when applied on $u$ and $v$, will produce either two real values or two categorical (integer) values.

- **Real-valued case:** $\mathbb{E}_h (h(u) \times h(v)) = \text{Kernel}(u, v)$

- **Categorical-valued case:** $\text{Pr}_h (h(u) = h(v)) = \text{Kernel}(u, v)$

For both cases, the inner products of hashed data approximate some (linear or nonlinear) kernel of the original data. With enough hashes, we can approach the kernel performance.

Next, we focus on a special hash function.
(0-Bit) Consistent Weighted Sampling: Procedure

This is a classical (but less known) result after many years of development in CS:

\[ \text{Input: } \text{Nonnegative data vector } u = (u_i \geq 0, i = 1 \text{ to } D) \]

For \( i \) from 1 to \( D \)

\[
\begin{align*}
    r_i & \sim \text{Gamma}(2, 1), \quad c_i \sim \text{Gamma}(2, 1), \quad \beta_i \sim \text{Uniform}(0, 1) \\
    t_i & \leftarrow \left\lfloor \frac{\log u_i}{r_i} + \beta_i \right\rfloor, \quad y_i \leftarrow \exp(r_i(t_i - \beta_i)), \quad a_i \leftarrow c_i/(y_i \exp(r_i))
\end{align*}
\]

\( i^* \leftarrow \text{arg min}_i a_i \)

Generate a new set of \( r_{ij}, c_{ij}, \beta_{ij} \) and apply the same procedure on \( u \) to obtain \( i^*_j \).

Apply the same procedure to all data vectors, using the same samples \( r_{ij}, c_{ij}, \beta_{ij} \).

For data vectors \( u \) and \( v \), we denote \( i_{u,j}^* \) and \( i_{v,j}^* \), \( j = 1, 2, \ldots, k \).
Apply CWS \((k)\) times on two nonnegative vectors \(u, v \in \mathbb{R}^D\). Then
\[
\Pr \{ u_{*,j} = v_{*,j} \} \approx K_{MM}(u, v)
\]
where
\[
K_{MM}(u, v) = \frac{\sum_{i=1}^{D} \min\{u_i, v_i\}}{\sum_{i=1}^{D} \max\{u_i, v_i\}}
\]

This provides a linear approximation to nonlinear min-max kernel.
0-Bit CWS for Building Learning Models

**LIBSVM input data format** (assuming binary data):

1 2:1 4:1 9:1 13:1 25:1
0 5:1 9:1 36:1
0 1:1 2:1 15:1 28:1
1 4:1 8:1 21:1 26:1 40:1

**0-bit CWS Samples for LIBSVM** (assuming $k = 3$):

\[
y_u \quad i^*_{u,1} : 1 \quad i^*_{u,2} : 1 \quad i^*_{u,3} : 1
\]
\[
y_v \quad i^*_{v,1} : 1 \quad i^*_{v,2} : 1 \quad i^*_{v,3} : 1
\]

**What if $i^*$ ranges too large?**: Simply store a few ($b$) bits only.
Generalized Min-Max Kernel

Consider the original data vector $u_i, i = 1$ to $D$. We define the following transformation, depending on whether an entry $u_i$ is positive or negative:

\[
\begin{align*}
\tilde{u}_{2i-1} &= u_i, \quad \tilde{u}_{2i} = 0 \quad \text{if } u_i > 0 \\
\tilde{u}_{2i-1} &= 0, \quad \tilde{u}_{2i} = -u_i \quad \text{if } u_i \leq 0
\end{align*}
\]

For example, when $D = 2$ and $u = [-5, 3]$, the transformed vector is $\tilde{u} = [0, 5, 3, 0]$.

\[
GMM(u, v) = \frac{\sum_{i=1}^{2D} \min(\tilde{u}_i, \tilde{v}_i)}{\sum_{i=1}^{2D} \max(\tilde{u}_i, \tilde{v}_i)}
\]

We name the corresponding hashing method the “generalized CWS” (GCWS) algorithm.
The radial basis function (RBF) kernel is widely used. For convenience, we recommend the following version:

\[ RBF(u, v; \gamma) = e^{-\gamma(1-\rho)} \]

where \( \rho = \rho(u, v) \) is the correlation and \( \gamma > 0 \) is a crucial tuning parameter. It is known that, if we sample \( w \sim \text{uniform}(0, 2\pi) \), \( r_i \sim N(0, 1) \) i.i.d., and let \( x = \sum_{i=1}^{D} u_i r_{ij} \), \( y = \sum_{i=1}^{D} v_i r_{ij} \), where \( \|u\|_2 = \|v\|_2 = 1 \), then we have

\[ E \left( \sqrt{2} \cos(\sqrt{\gamma}x + w) \sqrt{2} \cos(\sqrt{\gamma}y + w) \right) = e^{-\gamma(1-\rho)} \]

This provides a nice mechanism for linearizing the RBF kernel and the RFF method has become very popular in machine learning, computer vision, and beyond.
Kernel SVM Experiments

$l_2$-regularized kernel SVM (with parameter $C$) using libsvm pre-computed kernels (memory expensive). For linear kernel, we use LIBLINEAR (for the convenience of comparing with hashing results). For repeatability, the datasets are only of moderate sizes.
<table>
<thead>
<tr>
<th>Dataset</th>
<th># train</th>
<th># test</th>
<th># dim</th>
<th>linear (%)</th>
<th>RBF ($\gamma$) (%)</th>
<th>GMM (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DailySports</td>
<td>4560</td>
<td>4560</td>
<td>5625</td>
<td>77.7</td>
<td>97.6 (6)</td>
<td>99.6</td>
</tr>
<tr>
<td>Fourclass</td>
<td>431</td>
<td>431</td>
<td>2</td>
<td>72.6</td>
<td>81.2 (45)</td>
<td>100</td>
</tr>
<tr>
<td>IJCNN5k</td>
<td>5000</td>
<td>91701</td>
<td>22</td>
<td>91.9</td>
<td>95.8 (9)</td>
<td>95.1</td>
</tr>
<tr>
<td>Letter</td>
<td>15000</td>
<td>5000</td>
<td>16</td>
<td>61.7</td>
<td>97.4 (11)</td>
<td>97.3</td>
</tr>
<tr>
<td>MSD20k</td>
<td>20000</td>
<td>20000</td>
<td>90</td>
<td>66.7</td>
<td>68.0 (0.9)</td>
<td>71.1</td>
</tr>
<tr>
<td>Musk</td>
<td>3299</td>
<td>3299</td>
<td>166</td>
<td>95.1</td>
<td>99.3 (1.2)</td>
<td>99.2</td>
</tr>
<tr>
<td>PAMAP101</td>
<td>20000</td>
<td>20000</td>
<td>51</td>
<td>76.9</td>
<td>94.3 (1.5)</td>
<td>98.9</td>
</tr>
<tr>
<td>PAMAP102</td>
<td>20000</td>
<td>20000</td>
<td>51</td>
<td>81.2</td>
<td>95.7 (0.5)</td>
<td>98.8</td>
</tr>
<tr>
<td>PAMAP103</td>
<td>20000</td>
<td>20000</td>
<td>51</td>
<td>85.5</td>
<td>97.5 (0.5)</td>
<td>99.7</td>
</tr>
<tr>
<td>PAMAP104</td>
<td>20000</td>
<td>20000</td>
<td>51</td>
<td>84.0</td>
<td>97.3 (19)</td>
<td>99.3</td>
</tr>
<tr>
<td>PAMAP105</td>
<td>20000</td>
<td>20000</td>
<td>51</td>
<td>79.4</td>
<td>97.3 (18)</td>
<td>99.2</td>
</tr>
<tr>
<td>Satimage</td>
<td>2218</td>
<td>2217</td>
<td>36</td>
<td>80.5</td>
<td>90.5 (8)</td>
<td>91.5</td>
</tr>
<tr>
<td>Segment</td>
<td>1155</td>
<td>1155</td>
<td>19</td>
<td>92.7</td>
<td>97.0 (0.5)</td>
<td>97.8</td>
</tr>
<tr>
<td>SEMG1</td>
<td>900</td>
<td>900</td>
<td>3000</td>
<td>26.0</td>
<td>43.6 (4)</td>
<td>41.0</td>
</tr>
<tr>
<td>SEMG2</td>
<td>1800</td>
<td>1800</td>
<td>2500</td>
<td>19.3</td>
<td>29.0 (6)</td>
<td>54.0</td>
</tr>
<tr>
<td>SensIT20k</td>
<td>20000</td>
<td>19705</td>
<td>100</td>
<td>80.4</td>
<td>83.2 (0.1)</td>
<td>84.6</td>
</tr>
<tr>
<td>Sensorless</td>
<td>29255</td>
<td>29254</td>
<td>48</td>
<td>61.5</td>
<td>93.0 (0.4)</td>
<td>99.4</td>
</tr>
<tr>
<td>Shuttle500</td>
<td>500</td>
<td>43000</td>
<td>9</td>
<td>93.4</td>
<td>99.0 (90)</td>
<td>99.6</td>
</tr>
<tr>
<td>SVMGuide1</td>
<td>3089</td>
<td>4000</td>
<td>4</td>
<td>79.9</td>
<td>90.6 (50)</td>
<td>97.3</td>
</tr>
<tr>
<td>USPS</td>
<td>7291</td>
<td>2007</td>
<td>256</td>
<td>91.6</td>
<td>95.7 (4)</td>
<td>95.0</td>
</tr>
<tr>
<td>Vowel</td>
<td>528</td>
<td>462</td>
<td>10</td>
<td>27.9</td>
<td>48.1 (13)</td>
<td>62.8</td>
</tr>
<tr>
<td>IJCNN</td>
<td>49990</td>
<td>91701</td>
<td>22</td>
<td>92.6</td>
<td>– (9)</td>
<td>–</td>
</tr>
<tr>
<td>PAMAP101Large</td>
<td>186581</td>
<td>186580</td>
<td>51</td>
<td>79.2</td>
<td>– (1.5)</td>
<td>–</td>
</tr>
<tr>
<td>PAMAP105Large</td>
<td>185548</td>
<td>185548</td>
<td>51</td>
<td>83.4</td>
<td>– (18)</td>
<td>–</td>
</tr>
<tr>
<td>SensIT</td>
<td>78823</td>
<td>19705</td>
<td>100</td>
<td>80.5</td>
<td>– (0.1)</td>
<td>–</td>
</tr>
</tbody>
</table>

103
GCWS and RFF Hashing for Linearization of Nonlinear Kernels

**Solid**: Linearized GMM kernel using GCWS

**Dashed**: Linearized RBF kernel (at the best $\gamma$) using RFF

**Marked with ***: Linear method using original data

The original dimension is merely 16. Interestingly, with merely $k = 16$ samples, GCWS already produces better results than the original linear method.
Hashing Experiments on Larger Datasets

IJCNN: b = 8

PAMAP101Large: b = 8

Accuracy (%) vs. C for different k values.
The definition of the GMM kernel does not use any tuning parameters. One would expect the performance ought to be improved by introducing tuning parameters, for example

$$pGMM(u, v; \gamma) = \frac{\sum_{i=1}^{2D} (\min\{\tilde{u}_i, \tilde{v}_i\})^\gamma}{\sum_{i=1}^{2D} (\max\{\tilde{u}_i, \tilde{v}_i\})^\gamma}$$

$$epGMM(u, v; \gamma_1, \gamma_2) = e^{\gamma_2 \left( 1 - \frac{\sum_{i=1}^{2D} (\min\{\tilde{u}_i, \tilde{v}_i\})^{\gamma_1}}{\sum_{i=1}^{2D} (\max\{\tilde{u}_i, \tilde{v}_i\})^{\gamma_1}} \right)}$$
Comparisons with Deep Nets and Trees: M-Noise1, ..., M-Noise6

Comparisons with deep nets

Comparisons with boosted trees
Hashing Algorithms for Indexing and Efficient Near Neighbor Search

- **Problem:** Searching for “similar” objects is a basic operation in science and engineering. We aim at developing sublinear time algorithms for highly efficient near neighbor search.

- **Importance:** Owing to numerous applications, developing efficient algorithms for near neighbors has been an active topic of research since early days of modern computing.

- **Prior solutions:** As an example, minwise hashing is a standard tool used in search industry for approximate near neighbor search. One (among others) major limitation is the heavy preprocessing cost, which is both computational and energy intensive.
• **Our solutions:** For this particular example (i.e., minwise hashing), we have developed one permutation hashing and densified one permutation hashing.

**Ref:** P. Li, et. al. *One Permutation Hashing*, NIPS 2012

**Ref:** A. Shrivastava and P. Li, *Densifying One Permutation Hashing via Rotation for Fast Near Neighbor Search*, ICML 2014

**Ref:** A. Shrivastava and P. Li, *Improved Densification of One Permutation Hashing*, UAI 2014
Fast Near Neighbor Search by Hash Tables

**LSH:** Instead of scanning all data points to find the nearest neighbors (for an input query), we can partition the space into many bins by building hash tables.

<table>
<thead>
<tr>
<th>Index</th>
<th>Data Points</th>
<th>Index</th>
<th>Data Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00</td>
<td>8, 13, 251</td>
<td>00 00</td>
<td>2, 19, 83</td>
</tr>
<tr>
<td>00 01</td>
<td>5, 14, 19, 29</td>
<td>00 01</td>
<td>17, 36, 129</td>
</tr>
<tr>
<td>00 10</td>
<td>(empty)</td>
<td>00 10</td>
<td>4, 34, 52, 796</td>
</tr>
<tr>
<td>11 01</td>
<td>7, 24, 156</td>
<td>11 01</td>
<td>7, 198</td>
</tr>
<tr>
<td>11 10</td>
<td>33, 174, 3153</td>
<td>11 10</td>
<td>56, 989</td>
</tr>
<tr>
<td>11 11</td>
<td>61, 342</td>
<td>11 11</td>
<td>8, 9, 156, 879</td>
</tr>
</tbody>
</table>

For example, a table of $2^4 = 16$ partitions data into 16 bins. The point 8 is placed in bin 0000. To improve accuracy, we need to build many tables.

The key is how to place the data points into bins. Minwise hashing is one good method and there are many others including we have developed in the past: sign cauchy projections, sign stable projections, CoRE kernels, 0-bit consistent weighted sampling, etc.
Minwise Hashing: Notation

A binary (0/1) vector $\iff$ a set (locations of nonzeros).

Consider two sets $S_1, S_2 \subseteq \Omega = \{0, 1, 2, \ldots, D - 1\}$ (e.g., $D = 2^{64}$)

\[
f_1 = |S_1|, \quad f_2 = |S_2|, \quad a = |S_1 \cap S_2|.
\]

The resemblance $R$ is a popular measure of set similarity

\[
R = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|} = \frac{a}{f_1 + f_2 - a}.
\]

(Is it more rational than $\frac{a}{\sqrt{f_1 f_2}}$?)
Minwise Hashing: Standard Algorithm in the Context of Search

The standard practice in the search industry:

Suppose a random permutation $\pi$ is performed on $\Omega$, i.e.,

$$\pi : \Omega \rightarrow \Omega,$$

where $\Omega = \{0, 1, \ldots, D - 1\}$.

An elementary probability argument shows that

$$\Pr \left( \min(\pi(S_1)) = \min(\pi(S_2)) \right) = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|} = R.$$
An Example

\[ D = 5. \quad S_1 = \{0, 3, 4\}, \quad S_2 = \{1, 2, 3\}, \quad R = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|} = \frac{1}{5}. \]

One realization of the permutation \( \pi \) can be

\[
\begin{align*}
0 & \mapsto 3 \\
1 & \mapsto 2 \\
2 & \mapsto 0 \\
3 & \mapsto 4 \\
4 & \mapsto 1
\end{align*}
\]

\[ \pi(S_1) = \{3, 4, 1\} = \{1, 3, 4\}, \quad \pi(S_2) = \{2, 0, 4\} = \{0, 2, 4\} \]

In this example, \( \min(\pi(S_1)) \neq \min(\pi(S_2)) \).
Minwise Hashing in 0/1 Data Matrix

Original Data Matrix

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>S₁</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>S₂</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>S₃</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Permuted Data Matrix

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>π(S₁)</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>π(S₂)</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>π(S₃)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ \min(\pi(S₁)) = 2, \quad \min(\pi(S₂)) = 0, \quad \min(\pi(S₃)) = 0 \]
An Example with $k = 3$ Permutations

Input: sets $S_1$, $S_2$, ..., 

Hashed values for $S_1$ : 113 264 1091
Hashed values for $S_2$ : 2049 103 1091
Hashed values for $S_3$ : ...

.....
Minwise Hashing Estimator

After $k$ permutations, $\pi_1, \pi_2, \ldots, \pi_k$, one can estimate $R$ without bias:

$$\hat{R}_M = \frac{1}{k} \sum_{j=1}^{k} 1\{\min(\pi_j(S_1)) = \min(\pi_j(S_2))\},$$

$$\text{var}\left(\hat{R}_M\right) = \frac{1}{k} R (1 - R).$$
## Issues with Minwise Hashing and Our Solutions

1. **Expensive storage (and computation):** In the standard practice, each hashed value was stored using 64 bits.

   **Our solution:** b-bit minwise hashing by using only the lowest $b$ bits.

2. **How to do linear kernel learning:**

   **Our solution:** We show that b-bit minwise hashing results in positive definite (PD) linear kernel matrix. The data dimensionality is reduced from $2^{64}$ to $2^b$.

3. **Expensive and energy-consuming (pre)processing for $k$ permutations:**

   **Our solution:** One permutation hashing, which is even more accurate.
**Integrating b-Bit Minwise Hashing for (Linear) Learning**

**Very simple:**

1. Apply $k$ independent random permutations on each (binary) feature vector $x_i$ and store the lowest $b$ bits of each hashed value. The storage costs $nbk$ bits.

2. At run-time, expand a hashed data point into a $2^b \times k$-length vector, i.e. concatenate $k$ $2^b$-length vectors. The new feature vector has exactly $k$ 1’s.
An Example with $k = 3$ Permutations

Input: sets $S_1$, $S_2$, ...

Hashed values for $S_1$:

113  264  1091

Hashed values for $S_2$:

2049  103  1091

Hashed values for $S_3$: ...

....
Key observation: the estimator can be written as an inner product

\[
\hat{R}_M = \frac{1}{k'} \sum_{j=1}^{k} 1\{\min(\pi_j(S_1)) = \min(\pi_j(S_2))\}
\]

\[
= \frac{1}{k'} \sum_{j=1}^{k} \sum_{i=1}^{D} 1\{\min(\pi_j(S_1)) = i\} \times 1\{\min(\pi_j(S_2) = i\}
\]

(Recall the story of expanding the signs to be [01] or [10].)

The only issue is that \(D\) is too large. However, with \(b\)-bit minwise hashing, the space is only of size \(2^b\).
An Example with $k = 3$ Permutations and $b = 2$ Bits

For set (vector) $S_1$: (Original high-dimensional binary feature vector)

Hashed values : 113 264 1091
Binary : 1110001 10001000 100100001
Lowest $b = 2$ bits : 01 00 11
Decimal values : 1 0 3
Expansions ($2^b$) : 0100 1000 0001

New binary feature vector : $[0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1] \times \frac{1}{\sqrt{3}}$

Same procedures on sets $S_2, S_3, ...$
Experiments on Webspam (3-gram) Data: Testing Accuracy

- Dashed: using the original data (24GB disk space).

- Solid: $b$-bit hashing. Using $b = 8$ and $k = 200$ achieves about the same test accuracies as using the original data. Space: **70MB** ($350000 \times 200$)
What Is Happening?

1. **By engineering:**
   
   Webspam, unigram, 254 dim $\Rightarrow$ 3-gram, 16M dim, 4000 nonzeros per doc

   Accuracy: $93.3\% \Rightarrow 99.6\%$

2. **By probability/statistics:**

   16M dim, 4000 nonzeros $\Rightarrow k = 200$ nonzeros, $2^b \times k = 51200$ dim

   Accuracy: $99.6\% \Rightarrow 99.6\%$

Hashing can be viewed as part of feature engineering
spam: Accuracy

svm: k = 100
b = 1
b = 2
b = 4
b = 6, 8, 10, 16

C

Accuracy (%)

80
82
84
86
88
90
92
94
96
98
100

10^{-3} 10^{-2} 10^{-1} 10^0 10^1 10^2

spam: Accuracy

svm: k = 200
b = 1
b = 2
b = 4
b = 6, 8, 10, 16

C

Accuracy (%)

80
82
84
86
88
90
92
94
96
98
100

10^{-3} 10^{-2} 10^{-1} 10^0 10^1 10^2

spam: Accuracy

svm: k = 300
b = 4

C

Accuracy (%)

80
82
84
86
88
90
92
94
96
98
100

10^{-3} 10^{-2} 10^{-1} 10^0 10^1 10^2

spam: Accuracy

svm: k = 500
b = 1
b = 2
b = 4
b = 6, 8, 10, 16

C

Accuracy (%)

80
82
84
86
88
90
92
94
96
98
100

10^{-3} 10^{-2} 10^{-1} 10^0 10^1 10^2

spam: Accuracy
• They did not include data loading time (which is small for b-bit hashing)
• The original training time is about 200 seconds.
• b-bit minwise hashing needs about $3 \sim 7$ seconds ($3$ seconds when $b = 8$).
However, here we assume the test data have already been processed.
Comparison with Very Sparse Random Projections

8-bit minwise hashing (dashed, red): $k = 200$

Sparse random projections and variants (e.g., VW): $k = 10^4 \sim 10^6$. 

![Graph showing accuracy of SVM and Spam: Accuracy vs. k for different values of C (0.01, 0.1, 1) and b = 8 hashing, with y-axis Accuracy (%) and x-axis k ranging from $10^1$ to $10^6$.](image)
The Problem of Expensive Preprocessing

200 or 500 permutations (or even more for LSH) on the entire data can be very expensive. A serious issue when the new testing data have not been processed.

Three solutions:

1. **Parallel solution by GPUs**: Achieved up to 100-fold improvement in speed.
   
   **Ref**: Li, Shrivastava, König, *GPU-Based Minwise Hashing*, WWW’12 (poster)

2. **Conditional Random Sampling (CRS)**: (Useful for other applications)
   
   **Ref**: Li and Church, *Using Sketches to Estimate Associates*, EMNLP 2005
   
   **Ref**: Li, Church, Hastie *Conditional Random Sampling …*, NIPS 2006

3. **One Permutation Hashing**: (Recommended for learning and search)
   
   **Ref**: Li, Owen, Zhang, *One Permutation Hashing*, NIPS 2012
   
   **Ref**: Shrivastava and Li, *Densified One Permutation Hashing …*, ICML 2014
Intuition: Minwise Hashing Ought to Be Wasteful

**Original Data Matrix**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$:</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$S_2$:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$S_3$:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Permuted Data Matrix**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\pi(S_1)$:</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$\pi(S_2)$:</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$\pi(S_3)$:</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Only store the minimums and repeat the process $k$ (e.g., 500) times.
One Permutation Hashing

$S_1, S_2, S_3 \subseteq \Omega = \{0, 1, \ldots, 15\}$ (i.e., $D = 16$). The figure presents the permuted sets as three binary (0/1) vectors:

$\pi(S_1) = \{2, 4, 7, 13\}$, \hspace{1cm} $\pi(S_2) = \{0, 6, 13\}$, \hspace{1cm} $\pi(S_3) = \{0, 1, 10, 12\}$

One permutation hashing: divide the space $\Omega$ evenly into $k = 4$ bins and select the smallest nonzero in each bin.
Experimental Results on Webspam Data

One permutation hashing (zero coding) is even slightly more accurate than $k$-permutation hashing (at merely $1/k$ of the original cost).
Limitation of One Permutation Hashing

One permutation hashing can not be directly used for near neighbor search by building hash tables because empty bins do not offer indexing capability.

In other words, because of these empty bins, it is not possible to determine which bin value to use for bucketing.
<table>
<thead>
<tr>
<th>Index</th>
<th>Data Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00</td>
<td>8, 13, 251</td>
</tr>
<tr>
<td>00 01</td>
<td>5, 14, 19, 29</td>
</tr>
<tr>
<td>00 10</td>
<td>(empty)</td>
</tr>
<tr>
<td>11 01</td>
<td>7, 24, 156</td>
</tr>
<tr>
<td>11 10</td>
<td>33, 174, 3153</td>
</tr>
<tr>
<td>11 11</td>
<td>61, 342</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index</th>
<th>Data Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00</td>
<td>2, 19, 83</td>
</tr>
<tr>
<td>00 01</td>
<td>17, 36, 129</td>
</tr>
<tr>
<td>00 10</td>
<td>4, 34, 52, 796</td>
</tr>
<tr>
<td>11 01</td>
<td>7, 198</td>
</tr>
<tr>
<td>11 10</td>
<td>56, 989</td>
</tr>
<tr>
<td>11 11</td>
<td>8, 9, 156, 879</td>
</tr>
</tbody>
</table>
Neither Zero-coding nor Random-Coding Would Work

Zero-coding, or “empty-equal” (EE), scheme: If empty bins dominate, then two sparse vectors will become artificially “similar”.

Random-coding, or “empty-not-equal” (ENE), scheme: By coding an empty bin randomly, again if empty bins dominate, then two sparse vectors which are similar in terms of the original resemblance may artificially become not so similar.

Why zero-coding seems to work with linear learning? It works because in the worst case (when the number of bins is the same as the number of columns), we get back the original inner product (which is not necessarily bad).
Our Proposal: One Permutation with Rotation

The original one permutation hashing (OPH) is densified to become $H$:

$$
\pi(S_1) = [0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 1, 1, 0]
\pi(S_2) = [0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0]
$$

$$
\text{OPH}(\pi(S_1)) = [E, 1, E, 2, 0, 1] \quad \text{H}(\pi(S_1)) = [1+C, 1, 2+C, 2, 0, 1]
\text{OPH}(\pi(S_2)) = [E, 1, E, 0, 0, E] \quad \text{H}(\pi(S_2)) = [1+C, 1, C, 0, 0, 1+2C]
$$

$C \geq D/k + 1$ is a constant for avoiding undesired collision.

**Theorem:** $\Pr(\mathcal{H}_j(\pi(S_1)) = \mathcal{H}_j(\pi(S_2))) = R$

**Ref:** Shrivastava and Li, *Densifying One Permutation Hashing ...*, ICML’14

136
Our proposal matches the standard minwise hashing.
The images depict two graphs comparing recall with respect to the number of tables ($L$) for different datasets and parameters. The graphs show the performance of two methods: Minwise and Proposed.

**Left Graph:**
- Dataset: MNIST
- Parameters: $K=10$, $T_0=0.5$
- The graph shows the recall as $L$ increases from 10 to 30 tables.

**Right Graph:**
- Dataset: NEWS20
- Parameters: $K=10$, $T_0=0.5$
- The graph shows the recall as $L$ increases from 20 to 120 tables.

Both graphs illustrate how the proposed method performs compared to Minwise.
An Even Better Scheme for Densification

For each bin, we toss a coin to decide whether we fill the bin (if empty) using either the nearest circular left bin or the nearest circular right bin.

<table>
<thead>
<tr>
<th>Direction Bits (q)</th>
<th>Bin 0</th>
<th>Bin 1</th>
<th>Bin 2</th>
<th>Bin 3</th>
<th>Bin 4</th>
<th>Bin 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>( H^+(S_1) )</td>
<td>0+2C</td>
<td>1</td>
<td>-1+C</td>
<td>0</td>
<td>0</td>
<td>1+2C</td>
</tr>
<tr>
<td>( H^+(S_2) )</td>
<td>1</td>
<td>1</td>
<td>-1+C</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**Theorem** \( \Pr \left( H_j^+ (\pi(S_1)) = H_j^+ (\pi(S_2)) \right) = R \)

Both schemes (\( H \) and \( H^+ \)) are unbiased linear hash functions.

Need the variance analysis to see the advantage of the new scheme.
Variance Analysis for Both Densification Schemes

**Theorem:** Variance of the estimator based on existing scheme ($\mathcal{H}$)

\[
\text{Var}(\hat{R}) = \frac{R}{k} + A \frac{R}{k} + B \frac{R \tilde{R}}{k} - R^2
\]

\[
A = 2\mathbb{E} \left[ \frac{N_{emp}}{k - N_{emp} + 1} \right], \quad B = (k + 1)\mathbb{E} \left[ \frac{k - N_{emp} - 1}{k - N_{emp} + 1} \right]
\]

The distribution of $N_{emp}$ can be found in *(Ref: Li, Owen, Zhang, NIPS’12)*.

**Theorem:** Variance of the estimator based on improved scheme ($\mathcal{H}^+$)

\[
\text{Var}(\hat{R}^+) = \frac{R}{k} + A^+ \frac{R}{k^2} + B^+ \frac{R \tilde{R}}{k^2} - R^2
\]

\[
A^+ = \mathbb{E} \left[ \frac{N_{emp}(4k - N_{emp} + 1)}{2(k - N_{emp} + 1)} \right]
\]

\[
B^+ = \mathbb{E} \left[ \frac{2k^3 + N_{emp}^2 - N_{emp}(2k^2 + 2k + 1) - 2k}{2(k - N_{emp} + 1)} \right]
\]
Theorem \[ V_{ar}(\hat{R}) - V_{ar}(\hat{R}^+) = \mathbb{E} \left[ \frac{(N_{emp})(N_{emp} - 1)}{2k^2(k - N_{emp} + 1)} [R - R\tilde{R}] \right] \geq 0 \]

- Straight line represents MSE of conventional \( k \)-permutation minwise hashing
- With too many bins, one permutation hashing will eventually stop helping.
Minwise Hashing (MinHash) versus SimHash

Ref: A. Shrivastava and P. Li, In Defense of Minhash over Simhash, AISTATS 2014

- SimHash is based on sign random projections (SRP) and is very popular
- Interestingly MinHash can be significantly better for sparse data (not necessarily binary)
- SimHash approximates cosine similarity while MinHash approximates resemblance
- For binary data, theoretically (and empirically) MinHash is better than SimHash even when the retrieval results are evaluated using cosine similarity as the ground truth
- For non-binary sparse data, empirically, MinHash is still better than SimHash even when the retrieval results are evaluated using cosine similarity as the ground truth
## Experiments on Near Neighbor Search with Hash Tables

<table>
<thead>
<tr>
<th>Dataset</th>
<th># Query</th>
<th># Train</th>
<th># Dim</th>
</tr>
</thead>
<tbody>
<tr>
<td>MNIST</td>
<td>10,000</td>
<td>60,000</td>
<td>784</td>
</tr>
<tr>
<td>NEWS20</td>
<td>2,000</td>
<td>18,000</td>
<td>1,355,191</td>
</tr>
<tr>
<td>NYTIMES</td>
<td>5,000</td>
<td>100,000</td>
<td>102,660</td>
</tr>
<tr>
<td>RCV1</td>
<td>5,000</td>
<td>100,000</td>
<td>47,236</td>
</tr>
<tr>
<td>URL</td>
<td>5,000</td>
<td>90,000</td>
<td>3,231,958</td>
</tr>
<tr>
<td>WEBSPAM</td>
<td>5,000</td>
<td>100,000</td>
<td>16,609,143</td>
</tr>
</tbody>
</table>

- Build hash tables (by MinHash or SimHash) and place $N$ training points into the tables.
- Hash a query point, retrieve $n$ training points according to hash values.
- Out of these $n$ retrieved points, $m$ points are within top-$T$ true similar points.
- \[ \text{Recall} = \frac{m}{T}, \quad \text{Fraction retrieved} = \frac{n}{N} \]
Binarized data and evaluated using cosine similarity

Recall vs. Fraction Retrieved for MNIST:
- Top 1
- Top 10
- Top 20
- Top 100

Recall vs. Fraction Retrieved for NEWS20:
- Top 1
- Top 10
- Top 20
- Top 100

Recall vs. Fraction Retrieved for NYTIMES:
- Top 1
- Top 10
- Top 20
- Top 100
Binarized data and evaluated using cosine similarity
Non-binary sparse data and evaluated using cosine similarity

We still use binarized data for MinHash and evaluate it based on cosine of real-valued data.
Conclusion: Trees & Boosting

- Trees & boosting are popular machine learning tools in practice. Robust logitboost and abc-boost provide surprisingly substantial improvements for classification.

- In many (if not most) applications, trees should be the first tool to try. Typically the results are never bad, although sometimes other methods such as DNN are (slightly) better.

- Two examples of promising applications of trees are learning to rank and risk control.

- Inherent problems with trees: slow, large model, not suitable for many-level nominal categorical data. In many important industrial applications, those are serious limitations.

- Adaptive data binning simplifies tree implementation and speeds up training.
Conclusion: Hashing for Machine Learning

- Hashing methods first (probabilistically) transform the data, then apply linear (e.g., logistic regression) or nonlinear (DNN) learning algorithms on the transformed data. The process is very efficient (e.g., linear cost) and is naturally online & parallelizable.

- Hashing for dimension reduction (reducing model) and data reduction (reducing nonzeros)

- Hashing for feature expansion to achieve nonlinear effect. If used approximately, the results of hashing + logistic regression can be close to results of expensive DNN.

- In practice, hashing methods can be used to make simpler model (e.g., single machine model) or more complex models (for better accuracy), depending on the scenarios.
Conclusion: Hashing for Indexing and Near Neighbor Search

- Indexing is the basic step for search. (Efficient) Near neighbor search is a fundamental task since the very beginning of computer science, with numerous practical applications.

- Hashing provides a general framework for sublinear time near neighbor search. However, the details of particular hashing methods matter crucially for the performance.

- One “secret” is to make data sparse and then apply appropriate hashing methods.
General Principles

1. Do simple, useful work

2. Think critically and doubt everything

3. Details matter