1. **Recent progress in learning with trees & boosting** and why they are 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.

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

3. **Hashing algorithms for indexing and efficient near neighbor search**
Speaker’s Current Research

- Learning to rank (LTR) with trees & boosting for image search and web search
- Many other applications with trees & boosting
- Hashing algorithms for duplicate detections, semantic search, image indexing etc
- Hashing algorithms for building large-scale and complex models for ads CTR prediction
- Hashing algorithms for recommendation systems
- Data stream algorithms
- Other applications of big data beyond search, for example, bigdata finance
- Compressed sensing with heavy-tailed and/or sparse designs with applications in networks, databases, data streams, anomaly detection (e.g., DDoS), etc.

The content of this tutorial is solely based on published papers
References for Tutorial Materials

1. **Learning with boosting & 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.

2. **Hashing for large-scale statistical machine learning**
   - **Ref:** P. Li, et. al., *Hashing Algorithms for Large-Scale Learning*, NIPS 2011.
   - **Ref:** P. Li, et. al., *Sign Cauchy Random Projections and Chi-Square Kernels*, NIPS 2013.
   - **Ref:** P. Li, *CoRE Kernels*, UAI 2014.
   - **Ref:** P. Li, *0-Bit Consistent Weighted Sampling*, KDD 2015.
3. **Hashing for indexing and efficient approximate near neighbor search**

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

4. **Hashing for indexing and searching maximum inner products** *(no time to cover)*

   - **Ref:** A. Shrivastava and P. Li, *Asymmetric LSH (ALSH) for Sublinear Time Maximum Inner Product Search (MIPS)*, NIPS 2014
   - **Ref:** A. Shrivastava and P. Li, *Asymmetric Minwise Hashing for Indexing Binary Inner Products and Set Containment*, WWW 2015
   - **Ref:** A. Shrivastava and P. Li, *Improved Asymmetric Locality Sensitive Hashing (ALSH) for Maximum Inner Product Search (MIPS)*, UAI 2015
Recent Progress on Boosting and Trees
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

$$\hat{p}_k = \Pr \{ y = k | X \}, \quad k = 0, 1, \ldots, K - 1,$$

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

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:**

\[
F(x) = F^{(M)}(x) = \sum_{m=1}^{M} \rho_m h(x; \mathbf{a}_m),
\]
$h(x; \mathbf{a})$ 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.
Learning Logistic Regression by Maximum Likelihood

Seek $F_{i,k}$ to maximize the multinomial 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

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

\[ r_{i,k} = \begin{cases} 1 & \text{if } y_i = k \\ 0 & \text{otherwise} \end{cases} \]
Two Basic Optimization Methods for Maximum Likelihood

1. **Newton’s Method**
   Uses the first and second derivatives of the loss function.
   The method in LogitBoost.

2. **Gradient Descent**
   Only uses the first order derivative of the loss function.

MART used a creative combination of gradient descent and Newton’s method.
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: \( F_{i,k} = 0, \ p_{i,k} = \frac{1}{K}, \ k = 0 \text{ to } K - 1, \ i = 1 \text{ to } N \)

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

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

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

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

9: End
The Original MART 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: \{ $R_{j,k,m}$ \}_{j=1}^{J} = J$-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: $p_{i,k} = \exp(F_{i,k})/\sum_{s=0}^{K-1} \exp(F_{i,s}), \ k = 0 \ to \ K - 1, \ i = 1 \ to \ N$

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, \( z_{i,k} = z_{i,k} = \frac{r_{i,k} - p_{i,k}}{p_{i,k} (1 - p_{i,k})} \) may approach infinity.
The Numerical Issue in LoigtBoost

(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 essentially free of numerical problems.

It turns out, the numerical issue does not really exist, especially when the trees are not too large.
Tree-Splitting Using the Second-Order Information

**Feature values:** $x_i, i = 1$ to $N$. Assume $x_1 \leq x_2 \leq \ldots \leq x_N$.

**Weight values:** $w_i, i = 1$ to $N$. **Response values:** $z_i, i = 1$ to $N$.

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

$$Gain(s) = SE_T - (SE_L + SE_R)$$

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

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

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

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

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.
Update terminal node values

In LogitBoost:

\[
\frac{\sum_{\text{node}} z_{i,k} w_{i,k}}{\sum_{\text{node}} w_{i,k}} = \frac{\sum_{\text{node}} r_{i,k} - p_{i,k}}{\sum_{\text{node}} p_{i,k}(1 - p_{i,k})},
\]

which is the same as in MART.
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: $\{R_{j,k,m}\}^J_{j=1} = J$-terminal node regression tree from $\{r_{i,k} - p_{i,k}, \ x_i\}^N_{i=1}$,
   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: $F_{i,k} = F_{i,k} + \nu \sum_{j=1}^J \beta_{j,k,m} \mathbf{1}_{x_i \in R_{j,k,m}}$
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*
IJCNN1 Test Errors

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

Test misclassification error

- MART
- LibSVM
- Robust LogitBoost

Iterations
Forest100k Test Errors

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

Test misclassification error

- SVM
- MART

Robust LogitBoost

Iterations

Test misclassification error

0 5000 10000 3000 3500 4000 4500 5000
Forest521k Test Errors

Test misclassification error

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

- SVM
- MART
- Robust LogitBoost
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?

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

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:

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

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

which could be derived without imposing any constraints on \( F_k \).
Singularity of Hessian without Sum-to-zero Constraint

Without sum-to-zero constraint $\sum_{k=0}^{K-1} F_{i,k} = 0$:

$$\frac{\partial L_i}{\partial F_{i,k}} = -(r_{i,k} - p_{i,k}), \quad \frac{\partial^2 L_i}{\partial F_{i,k}^2} = p_{i,k}(1 - p_{i,k}).$$

For example, when $K = 3$.

$$\begin{vmatrix}
\frac{\partial^2 L_i}{\partial p_0^2} & \frac{\partial^2 L_i}{\partial p_0 p_1} & \frac{\partial^2 L_i}{\partial p_0 p_2} \\
\frac{\partial^2 L_i}{\partial p_1 p_0} & \frac{\partial^2 L_i}{\partial p_1^2} & \frac{\partial^2 L_i}{\partial p_1 p_2} \\
\frac{\partial^2 L_i}{\partial p_2 p_0} & \frac{\partial^2 L_i}{\partial p_2 p_1} & \frac{\partial^2 L_i}{\partial p_2^2}
\end{vmatrix} =
\begin{vmatrix}
p_0(1 - p_0) & -p_0 p_1 & -p_0 p_2 \\
-p_1 p_0 & p_1(1 - p_1) & -p_1 p_2 \\
-p_2 p_0 & -p_2 p_1 & p_2(1 - p_2)
\end{vmatrix} = 0$$
Diagonal Approximation

(Friedman et al 2000 and Friedman 2001) used diagonal approximation of Hessian:

\[ \frac{K - 1}{K} \begin{bmatrix} \frac{\partial^2 L_i}{\partial p_0^2} & \frac{\partial^2 L_i}{\partial p_1^2} \\ \frac{\partial^2 L_i}{\partial p_0^2} & \frac{\partial^2 L_i}{\partial p_2^2} \end{bmatrix} = \frac{K - 1}{K} \begin{bmatrix} p_0(1 - p_0) & \frac{\partial^2 L_i}{\partial p_2^2} \\ \frac{\partial^2 L_i}{\partial p_0^2} & p_1(1 - p_1) \end{bmatrix} \]
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

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

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

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

MART and LogitBoost used:

\[ \frac{\partial L_i}{\partial F_{i,k}} = -(r_{i,k} - p_{i,k}), \quad \frac{\partial^2 L_i}{\partial F_{i,k}^2} = p_{i,k}(1 - p_{i,k}). \]
Assume class 0 is the base class. The determinant of the Hessian is

$$\begin{vmatrix} \frac{\partial^2 L_i}{\partial p_1^2} & \frac{\partial^2 L_i}{\partial p_1 p_2} \\ \frac{\partial^2 L_i}{\partial p_2 p_1} & \frac{\partial^2 L_i}{\partial p_2^2} \end{vmatrix} =$$

$$\begin{vmatrix} p_0(1 - p_0) + p_1(1 - p_1) + 2p_0p_1 & p_0 - p_0^2 + p_0p_1 + p_0p_2 - p_1p_2 \\ p_0 - p_0^2 + p_0p_1 + p_0p_2 - p_1p_2 & p_0(1 - p_0) + p_2(1 - p_2) + 2p_0p_2 \end{vmatrix}$$

$$= p_0p_1 + p_0p_2 + p_1p_2 - p_0p_1^2 - p_0p_2^2 - p_1p_2^2 - p_2p_1^2 - p_1p_0^2 - p_2p_0^2 + 6p_0p_1p_2,$$

independent of the choice of the base class.

However, diagonal approximation appears to be a must when using trees. Thus the choice of base class matters.
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?

• We should choose the base class based on performance (training loss). How?

• (One Idea) Exhaustively search for all $K$ base classes and choose the base class that leads to the best performance (smallest training loss).
  
  – Computationally expensive (but not too bad, unless $K$ is really large)

  – Good performance can be achieved.

• Many other ideas.
\textbf{ABC-MART} = ABC-Boost + MART.

\textbf{ABC-LogitBoost} = ABC-Boost + Robust LogitBoost.
The Original MART 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: $\{R_{j,k,m}\}_{j=1}^{J}$ = $J$-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: $p_{i,k} = \frac{\exp(F_{i,k})}{\sum_{s=0}^{K-1} \exp(F_{i,s})}$, $k = 0$ to $K - 1$, $i = 1$ 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: \( \{ R_{j,k,m} \}_{j=1}^{J} = J \)-terminal node tree from \( \{ r_{i,k} - p_{i,k} - (r_{i,b} - p_{i,b}), \, x_{i} \}_{i=1}^{N} \)

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

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>60000</td>
</tr>
<tr>
<td>M-Basic</td>
<td>2058</td>
<td>1843</td>
<td>1723</td>
<td>1602</td>
<td>10993</td>
<td>50000</td>
</tr>
<tr>
<td>M-Rotate</td>
<td>7674</td>
<td>6634</td>
<td>6813</td>
<td>5959</td>
<td>26584</td>
<td>50000</td>
</tr>
<tr>
<td>M-Image</td>
<td>5821</td>
<td>4727</td>
<td>4703</td>
<td>4268</td>
<td>19353</td>
<td>50000</td>
</tr>
<tr>
<td>M-Rand</td>
<td>6577</td>
<td>5300</td>
<td>5020</td>
<td>4725</td>
<td>18189</td>
<td>50000</td>
</tr>
<tr>
<td>M-RotImg</td>
<td>24912</td>
<td>23072</td>
<td>22962</td>
<td>22343</td>
<td>33216</td>
<td>50000</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>5000</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

![Graph showing comparisons between different methods such as SVM, SAA-3, DBN-3, abc-logit, mart, logit, abc-mart, and abc-logit for different degrees of correlation. The x-axis represents the degree of correlation ranging from 1 to 6, and the y-axis represents the error rate percentage ranging from 0 to 40% for the left graph and 0 to 18% for the right graph. The graph illustrates the performance of each method across varying degrees of correlation.](image-url)
## More Comparisons with SVM and Deep Learning

<table>
<thead>
<tr>
<th></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>
Training Loss Vs Boosting Iterations

Training loss vs boosting iterations for different models:
- abc-logit
- abc-mart
- logit
- mart

Poker525k: J=20, ν=0.1
Test Errors Vs Boosting Iterations

Mnist10k: J = 20, \( \nu = 0.1 \)

Letter15k: J = 20, \( \nu = 0.1 \)

Test mis-classification errors

Boosting iterations

Test mis-classification errors

Boosting iterations
Letter4k: $J = 20, \nu = 0.1$

Letter2k: $J = 20, \nu = 0.1$
Covertype290k: $J=20$, $\nu=0.1$

Covertype145k: $J=20$, $\nu=0.1$

Poker525k: $J = 20$, $\nu = 0.1$

Poker275k: $J = 20$, $\nu = 0.1$
1. **Poker150k**: $J = 20, \nu = 0.1$
   - Boosting iterations vs. Test mis-classification errors
   - Curves for `mart`, `logit`, and `abc-mart` are shown.

2. **Poker100k**: $J = 20, \nu = 0.1$
   - Boosting iterations vs. Test mis-classification errors
   - Curves for `mart`, `logit`, and `abc-mart` are shown.

3. **Poker25kT1**: $J = 6, \nu = 0.1$
   - Boosting iterations vs. Test mis-classification errors
   - Curves for `abc-logit`, `mart`, and `logit` are shown.

4. **Poker25kT2**: $J = 6, \nu = 0.1$
   - Boosting iterations vs. Test mis-classification errors
   - Curves for `abc-logit`, `mart`, and `logit` are shown.
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>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\nu = 0.04$</td>
<td>$\nu = 0.06$</td>
<td>$\nu = 0.08$</td>
<td>$\nu = 0.1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$J = 4$</td>
<td>3356 3060</td>
<td>3329 3019</td>
<td>3318 2855</td>
<td>3326 2794</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$J = 6$</td>
<td>3185 2760</td>
<td>3093 2626</td>
<td>3129 2656</td>
<td>3217 2590</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$J = 8$</td>
<td>3049 2558</td>
<td>3054 2555</td>
<td>3054 2534</td>
<td>3035 2577</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$J = 10$</td>
<td>3020 2547</td>
<td>2973 2521</td>
<td>2990 2520</td>
<td>2978 2506</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$J = 12$</td>
<td>2927 2498</td>
<td>2917 2457</td>
<td>2945 2488</td>
<td>2907 2490</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$J = 14$</td>
<td>2925 2487</td>
<td>2901 2471</td>
<td>2877 2470</td>
<td>2884 2454</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$J = 16$</td>
<td>2899 2478</td>
<td>2893 2452</td>
<td>2873 2465</td>
<td>2860 2451</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$J = 18$</td>
<td>2857 2469</td>
<td>2880 2460</td>
<td>2870 2437</td>
<td>2855 2454</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$J = 20$</td>
<td>2833 2441</td>
<td>2834 2444</td>
<td>2834 2444</td>
<td>2815 2440</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$J = 24$</td>
<td>2840 2447</td>
<td>2827 2431</td>
<td>2801 2427</td>
<td>2784 2455</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$J = 30$</td>
<td>2826 2457</td>
<td>2822 2443</td>
<td>2828 2470</td>
<td>2807 2450</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$J = 40$</td>
<td>2837 2482</td>
<td>2809 2440</td>
<td>2836 2447</td>
<td>2782 2506</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$J = 50$</td>
<td>2813 2502</td>
<td>2826 2459</td>
<td>2824 2469</td>
<td>2786 2499</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$J$</td>
<td>logitboost</td>
<td>abc-logit</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----</td>
<td>------------</td>
<td>-----------</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\nu = 0.04$</td>
<td>2936 <strong>2630</strong></td>
<td>2970 <strong>2600</strong></td>
<td>2980 2535</td>
<td>3017 2522</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\nu = 0.06$</td>
<td>2710 <strong>2263</strong></td>
<td>2693 <strong>2252</strong></td>
<td>2710 2226</td>
<td>2711 2223</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\nu = 0.08$</td>
<td>2599 <strong>2159</strong></td>
<td>2619 <strong>2138</strong></td>
<td>2589 2120</td>
<td>2597 2143</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\nu = 0.1$</td>
<td>2553 <strong>2122</strong></td>
<td>2527 <strong>2118</strong></td>
<td>2516 2091</td>
<td>2500 2097</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$J = 10$</td>
<td>2472 <strong>2084</strong></td>
<td>2468 <strong>2090</strong></td>
<td>2468 2090</td>
<td>2464 2095</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$J = 12$</td>
<td>2451 <strong>2083</strong></td>
<td>2420 <strong>2094</strong></td>
<td>2432 2063</td>
<td>2419 2050</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$J = 14$</td>
<td>2424 <strong>2111</strong></td>
<td>2437 <strong>2114</strong></td>
<td>2393 2097</td>
<td>2395 2082</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$J = 16$</td>
<td>2399 <strong>2088</strong></td>
<td>2402 <strong>2087</strong></td>
<td>2389 2088</td>
<td>2380 2097</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$J = 18$</td>
<td>2388 <strong>2128</strong></td>
<td>2414 <strong>2112</strong></td>
<td>2411 2095</td>
<td>2381 2102</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$J = 20$</td>
<td>2442 <strong>2174</strong></td>
<td>2415 <strong>2147</strong></td>
<td>2417 2129</td>
<td>2419 2138</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$J = 24$</td>
<td>2468 <strong>2235</strong></td>
<td>2434 <strong>2237</strong></td>
<td>2423 2221</td>
<td>2449 2177</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$J = 30$</td>
<td>2551 <strong>2310</strong></td>
<td>2509 <strong>2284</strong></td>
<td>2518 2257</td>
<td>2531 2260</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$J = 40$</td>
<td>2612 <strong>2353</strong></td>
<td>2622 <strong>2359</strong></td>
<td>2579 2332</td>
<td>2570 2341</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Mnist10k: $J = 16$, $\nu = 0.04$

Mnist10k: $J = 16$, $\nu = 0.06$

Mnist10k: $J = 16$, $\nu = 0.1$

Mnist10k: $J = 18$, $\nu = 0.04$

Mnist10k: $J = 18$, $\nu = 0.06$

Mnist10k: $J = 18$, $\nu = 0.1$

Mnist10k: $J = 20$, $\nu = 0.04$

Mnist10k: $J = 20$, $\nu = 0.06$

Mnist10k: $J = 20$, $\nu = 0.1$
Mnist10k: $J = 24$, $\nu = 0.04$

Mnist10k: $J = 24$, $\nu = 0.06$

Mnist10k: $J = 24$, $\nu = 0.1$

Mnist10k: $J = 30$, $\nu = 0.04$

Mnist10k: $J = 30$, $\nu = 0.06$

Mnist10k: $J = 30$, $\nu = 0.1$

Mnist10k: $J = 40$, $\nu = 0.04$

Mnist10k: $J = 40$, $\nu = 0.06$

Mnist10k: $J = 40$, $\nu = 0.1$
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></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>
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
Outline

1. Challenges in search & learning with high-dimensional data

2. Sign Cauchy random projections and $\chi^2$ kernels
   Ref: P. Li, G. Samorodnitsky, and J. Hopcroft, NIPS 2013

3. 0-bit Consistent weighted sampling
   Ref: P. Li, KDD 2015

4. CoRE Kernels
   Ref: P. Li, UAI 2014

5. b-bit minwise hashing and its applications in large-scale learning
   Ref: Li and Konig, Research Highlight Article in Commu. of the ACM, 2011
   Ref: Li, Shrivastava, Moore, and Konig, NIPS, 2011

6. One permutation hashing.  Ref: Li, Owen, and Zhang, NIPS 2012
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>Method</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.
**Another Source of High-Dimensional Data: Global Expansion**

**MNIST: handwritten digit classification data**

<table>
<thead>
<tr>
<th></th>
<th>Dim.</th>
<th>Time</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original + Linear SVM</td>
<td>768</td>
<td>80 sec</td>
<td>92.1%</td>
</tr>
<tr>
<td>Pairwise + Linear SVM</td>
<td>295,296</td>
<td>400 sec</td>
<td>98.4%</td>
</tr>
<tr>
<td>Original + Kernel SVM</td>
<td>768</td>
<td>Many Hours</td>
<td>98.6%</td>
</tr>
</tbody>
</table>
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).
1. Challenges in search & learning with high-dimensional data.

2. Sign Cauchy random projections and $\chi^2$ kernels.

3. $b$-bit minwise hashing and its applications in large-scale learning

4. One permutation hashing.

5. Densified one permutation hashing for fast near neighbor search

6. Others: CoRE Kernels, etc.
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

$l_2$-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:

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

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

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

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**

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

\[ A \times R = B \]

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

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

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

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

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

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

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

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

Sign Cauchy random projections & expansions:

\[
x = u \times R : \begin{bmatrix} -61.83 & 2.45 & 13.83 & -1322.05 \end{bmatrix}
\]
\[
\text{sgn}(x) : \begin{bmatrix} -1 & +1 & +1 & -1 \end{bmatrix}
\]
\[
\text{Expansion} : \begin{bmatrix} 01 & 10 & 10 & 01 \end{bmatrix}
\]

\[
y = v \times R : \begin{bmatrix} -11.64 & 936.91 & -343.43 & -12.45 \end{bmatrix}
\]
\[
\text{sgn}(y) : \begin{bmatrix} -1 & +1 & -1 & -1 \end{bmatrix}
\]
\[
\text{Expansion} : \begin{bmatrix} 01 & 10 & 01 & 01 \end{bmatrix}
\]

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

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

Expansion : 01 10 10 01

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

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**
“negative sign” $\Rightarrow$ “01”,  
“positive sign” $\Rightarrow$ “10”.

$k$ Cauchy projections $\Rightarrow$ 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
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:** \( E_h (h(u) \times h(v)) = \text{Kernel}(u, v) \)
- **Categorical-valued case:** \( \Pr_h (h(u) = h(v)) = \text{Kernel}(u, v) \)

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:

- **Ref:** P. Li, *0-Bit Consistent Weighted Sampling*, KDD 2015
(0-Bit) Consistent Weighted Sampling: Procedure

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

———

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

**Output:** Consistent uniform sample \((i^*, t^*)\)

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

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

(note that \( t^* \) is theoretically unbounded)

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

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 \((i^*_{u,j}, t^*_{u,j})\) and \((i^*_{v,j}, t^*_{v,j})\), \( j = 1, 2, \ldots, k \)
Apply CWS ($k$ times) on two nonnegative vectors $u, v \in \mathbb{R}^D$. Then

$$\Pr\left\{ (i_{u,j}^*, t_{u,j}^*) = (i_{v,j}^*, t_{v,j}^*) \right\} = K_{MM}(u, v)$$

where

**Min-Max kernel**: $$K_{MM}(u, v) = \frac{\sum_{i=1}^{D} \min\{u_i, v_i\}}{\sum_{i=1}^{D} \max\{u_i, v_i\}}$$

**0-bit CWS** is a significant simplification by discarding $t^*$ entirely:

$$\Pr\left\{ i_{u,j}^* = i_{v,j}^* \right\} \approx \Pr\left\{ (i_{u,j}^*, t_{u,j}^*) = (i_{v,j}^*, t_{v,j}^*) \right\} = K_{MM}(u, v)$$

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

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

<table>
<thead>
<tr>
<th>1</th>
<th>2:1</th>
<th>4:1</th>
<th>9:1</th>
<th>13:1</th>
<th>25:1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5:1</td>
<td>9:1</td>
<td>36:1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1:1</td>
<td>2:1</td>
<td>15:1</td>
<td>28:1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4:1</td>
<td>8:1</td>
<td>21:1</td>
<td>26:1</td>
<td>40:1</td>
</tr>
</tbody>
</table>

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

$y_u \ i_{u,1}^* : 1 \ i_{u,2}^* : 1 \ i_{u,3}^* : 1$

$y_v \ i_{v,1}^* : 1 \ i_{v,2}^* : 1 \ i_{v,3}^* : 1$

**What if $i^*$ ranges too large?:** Simply store a few ($b$) bits only.
\( l_2 \)-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.
Classification Experiments Using 0-bit CWS

CWS samples: \((i^*, t^*)\). 0-bit CWS samples: \((i^*)\). For building large-scale linear classifiers, we further reduce the space by using only \(b_i\) bits of \(i^*\). (Ref: Li et.al. NIPS 2011)

Left to right: we store only \(b_i = 1, 2, 4, 8\) bits for each sample \(i^*\).
Experiment on Original (Larger) Webspam (1-gram) Dataset

Applying the method to larger data is straightforward, except that we would not be able to compute the exact classification result of min-max kernel.

Original WebspamN1 (1-gram): 350,000 examples; Linear SVM accuracy is about 93%. The proposed 0-bit CWS can achieve $> 98\%$ accuracies given enough samples.
CoRE (Correlation-REsemblance) Kernels

CoRE Kernel:

\[
\left[ \frac{\sum_{i=1}^{D} u_i v_i}{\sqrt{\sum_{i=1}^{D} u_i^2 \sum_{i=1}^{D} v_i^2}} \right] \times \left[ \frac{\sum_{i=1}^{D} 1\{u_i \neq 0\} \text{ and } 1\{v_i \neq 0\}}{\sum_{i=1}^{D} 1\{u_i \neq 0\} \text{ or } 1\{v_i \neq 0\}} \right]
\]

Classification accuracies (%) using a variety of kernels

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Linear</th>
<th>Linear (Binary)</th>
<th>Resemblance</th>
<th>CoRE</th>
</tr>
</thead>
<tbody>
<tr>
<td>M-Basic</td>
<td>90.0</td>
<td>88.9</td>
<td>95.9</td>
<td>97.0</td>
</tr>
<tr>
<td>MNIST10k</td>
<td>90.0</td>
<td>88.8</td>
<td>95.5</td>
<td>96.6</td>
</tr>
<tr>
<td>M-Rotate</td>
<td>48.0</td>
<td>44.4</td>
<td>80.3</td>
<td>87.6</td>
</tr>
<tr>
<td>RCV1</td>
<td>96.3</td>
<td>95.6</td>
<td>96.5</td>
<td>97.0</td>
</tr>
<tr>
<td>USPS</td>
<td>91.8</td>
<td>87.4</td>
<td>92.5</td>
<td>95.5</td>
</tr>
<tr>
<td>Youtube</td>
<td>47.6</td>
<td>46.5</td>
<td>51.1</td>
<td>53.1</td>
</tr>
</tbody>
</table>
Transforming CoRE Kernels into Linear Kernels

Basic Idea: $b$-bit minwise hashing + $k$ random projections

For M-Rotate dataset, usual random projections (and variants), which approximate correlations, can at most achieve 48%.
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
**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.

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 $\leftrightarrow$ 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.,

$$\pi : \Omega \longrightarrow \Omega,$$

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

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

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

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>$S_1$</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>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>$S_3$</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>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>$\pi(S_1)$</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>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

$\min(\pi(S_1)) = 2, \ \min(\pi(S_2)) = 0, \ \min(\pi(S_3)) = 0$
An Example with \( k = 3 \) Permutations

Input: sets \( S_1, S_2, \ldots \),

- 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, $\pi_1, \pi_2, ..., \pi_k$, 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 $2^b$. 

109
An Example with $k = 3$ Permutations and $b = 2$ Bits

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

<table>
<thead>
<tr>
<th>Hashed values</th>
<th>113</th>
<th>264</th>
<th>1091</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary</td>
<td>1110001</td>
<td>100001000</td>
<td>10001000011</td>
</tr>
<tr>
<td>Lowest $b = 2$ bits</td>
<td>01</td>
<td>00</td>
<td>11</td>
</tr>
<tr>
<td>Decimal values</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Expansions ($2^b$)</td>
<td>0100</td>
<td>1000</td>
<td>0001</td>
</tr>
</tbody>
</table>

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 $\implies$ 3-gram, 16M dim, 4000 nonzeros per doc

   Accuracy: 93.3% $\implies$ 99.6%

2. **By probability/statistics:**

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

   Accuracy: 99.6% $\implies$ 99.6%

Hashing can be viewed as part of feature engineering
• 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 ∼ 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 vs. k for SVM: VW vs b = 8 hashing and Spam: Accuracy]
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>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>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</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>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>
<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>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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

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

\[ \pi(S_1) = \{2, 4, 7, 13\}, \quad \pi(S_2) = \{0, 6, 13\}, \quad \pi(S_3) = \{0, 1, 10, 12\} \]

\[
\begin{array}{cccccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\
\pi(S_1): & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\pi(S_2): & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
\pi(S_3): & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
\end{array}
\]

**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>

122
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 $H$:

\[
\begin{align*}
\pi(S_1) &\quad 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 & 21 & 22 & 23 \\
\pi(S_1) & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\
\pi(S_2) & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0
\end{align*}
\]

\[
\begin{align*}
\text{OPH}(\pi(S_1)) &= [E, 1, E, 2, 0, 1] & \text{H}(\pi(S_1)) &= [1+C, 1, 2+C, 2, 0, 1] \\
\text{OPH}(\pi(S_1)) &= [E, 1, E, 0, 0, E] & \text{H}(\pi(S_1)) &= [1+C, 1, C, 0, 0, 1+2C]
\end{align*}
\]

$C \geq D/k + 1$ 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
Our proposal matches the standard minwise hashing.
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>1+C</td>
<td>1</td>
<td>1+C</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>(H'(S_2))</td>
<td>0+2C</td>
<td>1</td>
<td>1+C</td>
<td>0</td>
<td>0</td>
<td>1+2C</td>
</tr>
</tbody>
</table>

**Theorem** \(\Pr(H'_j^+(\pi(S_1)) = H'_j^+(\pi(S_2))) = R\)

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 \((\mathcal{H})\)

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

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

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

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

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

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

\[
B^+ = \mathbb{E}\left[\frac{2k^3 + N_{emp}^2 - N_{emp}(2k^2 + 2k + 1) - 2k}{2(k - N_{emp} + 1)}\right]
\]
Experimental Study for Verifying Variance Analysis

**Theorem** \[ Var(\hat{R}) - Var(\hat{R}^+) = \mathbb{E} \left[ \frac{(N_{emp})(N_{emp} - 1)}{2k^2(k - N_{emp} + 1)} [R - \hat{R}] \right] \geq 0 \]

![Graph showing MSE vs. k (Num of Hashes)]

- 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.
- Recall $= m/T$, Fraction retrieved $= n/N$
Binarized data and evaluated using cosine similarity

- MNIST: Top 1
- MNIST: Top 10
- MNIST: Top 20
- MNIST: Top 100
- NEWS20: Top 1
- NEWS20: Top 10
- NEWS20: Top 20
- NEWS20: Top 100
- NYTIMES: Top 1
- NYTIMES: Top 10
- NYTIMES: Top 20
- NYTIMES: Top 100
Binarized data and evaluated using cosine similarity

RCV1: Top 1
- SimHash
- MinHash

RCV1: Top 10
- SimHash
- MinHash

RCV1: Top 20
- SimHash
- MinHash

RCV1: Top 100
- SimHash
- MinHash

URL: Top 1
- SimHash
- MinHash

URL: Top 10
- SimHash
- MinHash

URL: Top 20
- SimHash
- MinHash

URL: Top 100
- SimHash
- MinHash

WEBSPAM: Top 1
- SimHash
- MinHash

WEBSPAM: Top 10
- SimHash
- MinHash

WEBSPAM: Top 20
- SimHash
- MinHash

WEBSPAM: Top 100
- SimHash
- MinHash
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: very slow, large model, not suitable for many-level nominal categorical data. In many important industrial applications, those are serious limitations.
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 “secrete” 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