THIS IS YOUR MACHINE LEARNING SYSTEM?

YUP! YOU POUR THE DATA INTO THIS BIG PILE OF LINEAR ALGEBRA, THEN COLLECT THE ANSWERS ON THE OTHER SIDE.

WHAT IF THE ANSWERS ARE WRONG?

JUST STIR THE PILE UNTIL THEY START LOOKING RIGHT.
Matrix Factorization
What is MF and what can you do with it?
Recovering latent factors in a matrix

\[ \begin{array}{cccc}
 v_{11} & \ldots & \\
 \ldots & \ldots & \\
 \ldots & \ldots & \\
 v_{ij} & & \\
 \ldots & & \\
 v_{nm} & & \\
\end{array} \]

\( m \) columns

\( n \) rows
Recovering latent factors in a matrix

\[
\begin{bmatrix}
x_1 & y_1 \\
x_2 & y_2 \\
.. & .. \\
.. & .. \\
x_n & y_n \\
\end{bmatrix}
\times
\begin{bmatrix}
a_1 & a_2 & .. & .. & a_m \\
b_1 & b_2 & .. & .. & b_m \\
.. & .. & .. & .. & .. \\
.. & .. & .. & .. & .. \\
v_{11} & .. & .. & .. & v_{nm} \\
\end{bmatrix}
\approx
\begin{bmatrix}
v_{11} & .. & .. & .. & v_{nm} \\
.. & .. & .. & .. & .. \\
.. & .. & .. & .. & .. \\
.. & .. & .. & .. & .. \\
\end{bmatrix}
\]
What is this for?
MF for collaborative filtering
What is collaborative filtering?

Help us make better recommendations. You can refine your recommendations by rating items or adjusting the checkboxes.

Items you've purchased

- In the Heart of the Sea: The Tragedy of the Whaleship Essex
  - Nathaniel Philbrick

- On Tyranny: Twenty Lessons from the Twentieth Century
  - Timothy Snyder

- Behave: The Biology of Humans at Our Best and Worst
  - Robert M. Sapolsky

Your Rating:

- Rate this item
- This was a gift
- Don’t use for recommendations

Need Help?
Visit our help area to learn more.
Recovering latent factors in a matrix

$V[i,j] = \text{user } i\text{'s rating of movie } j$

$v_{11}$ ...
...
$v_{ij}$ ...
...
$v_{nm}$
Recovering latent factors in a matrix

\[ V[i,j] = \text{user i's rating of movie j} \]
Semantic Factors (Koren et al., 2009)
Semantic Factors (Koren et al., 2009)
Recovering latent factors in a matrix

$n$ users

\[
\begin{pmatrix}
  x_1 & y_1 \\
  x_2 & y_2 \\
  \vdots & \vdots \\
  x_n & y_n \\
\end{pmatrix}
\]

$m$ movies

\[
\begin{pmatrix}
  a_1 & a_2 & \ldots & \ldots & a_m \\
  b_1 & b_2 & \ldots & \ldots & b_m \\
\end{pmatrix}
\]

$V[i,j] = \text{user } i\text{'s rating of movie } j$

$V[i,j] \approx \begin{pmatrix}
  v_{11} & \ldots \\
  \vdots & \vdots \\
  v_{ij} & \ldots \\
  \vdots & \vdots \\
  \vdots & \vdots \\
  v_{nm} & \\
\end{pmatrix}$
MF for image modeling
MF for images

2 prototypes

10,000 pixels

1000 * 10,000,00

V[i,j] = pixel j in image i

x1 y1
x2 y2
...

a1 a2 ... am
b1 b2 ... bm

v11 ...
...
vmn
MF for modeling text
Recovering latent factors in a matrix

\[ V[i,j] = \text{TFIDF score of term } j \text{ in doc } i \]
Recovering latent factors in a matrix

- The Neatest Little Guide to Stock Market Investing
- Investing For Dummies, 4th Edition
- The Little Book of Common Sense Investing: The Only Way to Guarantee Your Fair Share of Stock Market Returns
- The Little Book of Value Investing
- Value Investing: From Graham to Buffett and Beyond
- Rich Dad’s Guide to Investing: What the Rich Invest in, That the Poor and the Middle Class Do Not!
- Investing in Real Estate, 5th Edition
- Stock Investing For Dummies
- Rich Dad’s Advisors: The ABC’s of Real Estate Investing: The Secrets of Finding Hidden Profits Most Investors Miss

\[ [v_{ij}] = \text{TFIDF score of term } j \text{ in doc } i \]

Recovering latent factors in a matrix

doc term matrix

V[i,j] = TFIDF score of term j in doc i

dummy stock
saving advice ...

m terms

x1 y1
x2 y2
.. ..

.. ..

xn yn

a1 a2 .. ... am
b1 b2 .. ... bm

20
MF vs other learning tasks
MF is like linear regression

\[ y = \alpha + \beta x, \]
MF is like multiple-output multi-variable linear regression

\[ y_1 = x \cdot w_1 \]
\[ y_2 = x \cdot w_2 \]
\[ \ldots \]
\[ y_m = x \cdot w_m \]
Multi-output linear regression as MF

**Examples**

- $x_{11}$ $x_{12}$
- $x_{21}$ $x_{22}$
- ...
- $x_{n1}$ $y_n$

$X$

**$m$ weight vectors**

- $a_1$ $a_2$ $W$ ...
- $b_1$ $b_2$ ...
- $b_m$

$m$ outputs for each $x_i$

- $v_{11}$ ...
- ...
- $v_{nm}$

$Y$

$y_1 = x \cdot w_1$

$y_2 = x \cdot w_2$

$...$

$y_m = x \cdot w_m$
MF is like clustering
**k-means Clustering**

Each point is in one cluster.

Each cluster is a weighted average of points.

[Diagram showing clusters and centroids]
**k-means as MF**

*indicators for r clusters*

\[
\begin{pmatrix}
0 & 1 \\
1 & 0 \\
\vdots & \vdots \\
x_n & y_n
\end{pmatrix}
\]

*cluster means*

\[
\begin{pmatrix}
a_1 & a_2 & \cdots & a_m \\
b_1 & b_2 & \cdots & b_m
\end{pmatrix}
\]

*original data set*

\[
\begin{pmatrix}
v_{11} & \cdots \\
\vdots & \vdots \\
v_{ij} & \cdots \\
v_{nm}
\end{pmatrix}
\]
MF is “soft” clustering – each example is a weighted sum of clusters.

\[
\begin{pmatrix}
  x_1 & y_1 \\
  x_2 & y_2 \\
  \vdots & \vdots \\
  x_n & y_n \\
\end{pmatrix}
\times
\begin{pmatrix}
  a_1 & a_2 & \ldots & \ldots & a_m \\
  b_1 & b_2 & \ldots & \ldots & b_m \\
\end{pmatrix}
\approx
\begin{pmatrix}
  v_{11} & \ldots \\
  \vdots & \ddots \\
  \vdots & \ldots \\
  v_{nm} \\
\end{pmatrix}
\]
How do you do MF?
Large-Scale Matrix Factorization with Distributed Stochastic Gradient Descent

Rainer Gemulla

talk pilfered from

Peter J. Haas    Yannis Sismanis    Erik Nijkamp
Collaborative Filtering

- **Problem**
  - Set of users
  - Set of items (movies, books, jokes, products, stories, ...)
  - Feedback (ratings, purchase, click-through, tags, ...)

- **Predict additional items a user may like**
  - Assumption: Similar feedback $\implies$ Similar taste

- **Example**

<table>
<thead>
<tr>
<th></th>
<th>Avatar</th>
<th>The Matrix</th>
<th>Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>?</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Bob</td>
<td>3</td>
<td>2</td>
<td>?</td>
</tr>
<tr>
<td>Charlie</td>
<td>5</td>
<td>?</td>
<td>3</td>
</tr>
</tbody>
</table>

- Netflix competition: 500k users, 20k movies, 100M movie ratings, 3M question marks
Recovering latent factors in a matrix

\[ r \]

\[ \begin{pmatrix} x_1 & y_1 \\ x_2 & y_2 \\ \vdots & \vdots \\ x_n & y_n \end{pmatrix} \]

\[ W \]

\[ \begin{pmatrix} a_1 & a_2 & \cdots & a_m \\ b_1 & b_2 & \cdots & b_m \end{pmatrix} \]

\[ H \]

\[ \approx \]

\[ \begin{pmatrix} v_{11} & \cdots \\ \vdots & \ddots \\ v_{ij} & \cdots \\ \vdots & \ddots \end{pmatrix} \]

\[ V \]

\[ V[i,j] = \text{user i's rating of movie j} \]
Semantic Factors (Koren et al., 2009)
Latent Factor Models

- Discover latent factors ($r = 1$)

<table>
<thead>
<tr>
<th></th>
<th>Avatar</th>
<th>The Matrix</th>
<th>Up</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(2.24)</td>
<td>(1.92)</td>
<td>(1.18)</td>
</tr>
<tr>
<td>Alice</td>
<td>4</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>(1.98)</td>
<td>(3.8)</td>
<td>(2.3)</td>
<td></td>
</tr>
<tr>
<td>Bob</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>(1.21)</td>
<td>(2.7)</td>
<td>(2.3)</td>
<td></td>
</tr>
<tr>
<td>Charlie</td>
<td>5</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>(2.30)</td>
<td>(5.2)</td>
<td>(2.7)</td>
<td></td>
</tr>
</tbody>
</table>

- Minimum loss

$$\min_{W,H} \sum_{(i,j) \in Z} (V_{ij} - [WH]_{ij})^2$$
Latent Factor Models

Discover latent factors ($r = 1$)

<table>
<thead>
<tr>
<th>Avatar</th>
<th>The Matrix</th>
<th>Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2.24)</td>
<td>(1.92)</td>
<td>(1.18)</td>
</tr>
<tr>
<td>Alice</td>
<td>?</td>
<td>4</td>
</tr>
<tr>
<td>(1.98)</td>
<td>(4.4)</td>
<td>(3.8)</td>
</tr>
<tr>
<td>Bob</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>(1.21)</td>
<td>(2.7)</td>
<td>(2.3)</td>
</tr>
<tr>
<td>Charlie</td>
<td>5</td>
<td>?</td>
</tr>
<tr>
<td>(2.30)</td>
<td>(5.2)</td>
<td>(4.4)</td>
</tr>
</tbody>
</table>

Minimum loss

$$\min_{W,H,u,m} \sum_{(i,j) \in Z} (V_{ij} - \mu - u_i - m_j - [WH]_{ij})^2$$

$$+ \lambda (\|W\| + \|H\| + \|u\| + \|m\|)$$

Bias, regularization
Matrix factorization as SGD

require that the loss can be written as

\[ L = \sum_{(i,j) \in Z} l(V_{ij}, W_{i*}, H_{*j}) \]

Algorithm 1 SGD for Matrix Factorization

Require: A training set \( Z \), initial values \( W_0 \) and \( H_0 \)

while not converged do

   Select a training point \( (i, j) \in Z \) uniformly at random.

   \[ W_{i*}' \leftarrow W_{i*} - \epsilon_n N \frac{\partial}{\partial W_{i*}} l(V_{ij}, W_{i*}, H_{*j}) \]

   \[ H_{*j}' \leftarrow H_{*j} - \epsilon_n N \frac{\partial}{\partial H_{*j}} l(V_{ij}, W_{i*}, H_{*j}) \]

end while

why does this work?
Matrix factorization as SGD - why does this work? Here’s the key claim:

require that the loss can be written as

\[ L = \sum_{(i,j) \in Z} l(V_{ij}, W_{i*}, H_{*j}) \]

\[
\frac{\partial}{\partial W_{i'k}} L_{ij}(W, H) = \begin{cases} 
0 & \text{if } i \neq i' \\
\frac{\partial}{\partial W_{ik}} l(V_{ij}, W_{i*}, H_{*j}) & \text{otherwise}
\end{cases}
\]

\[
\frac{\partial}{\partial H_{kj'}} L_{ij}(W, H) = \begin{cases} 
0 & \text{if } j \neq j' \\
\frac{\partial}{\partial H_{kj}} l(V_{ij}, W_{i*}, H_{*j}) & \text{otherwise}
\end{cases}
\]
Checking the claim

\[ \frac{\partial}{\partial W_{i*}} L(W, H) = \frac{\partial}{\partial W_{i*}} \sum_{(i',j) \in Z} L_{i'j}(W_{i*}, H_{*j}) = \sum_{j \in Z_{i*}} \frac{\partial}{\partial W_{i*}} L_{ij}(W_{i*}, H_{*j}), \]

where \( Z_{i*} = \{ j : (i, j) \in Z \} \).

\[ \frac{\partial}{\partial H_{*j}} L(W, H) = \sum_{i \in Z_{*j}} \frac{\partial}{\partial W_{*j}} L_{ij}(W_{i*}, H_{*j}), \]

where \( Z_{*j} = \{ i : (i, j) \in Z \} \).

Think for SGD for logistic regression

- LR loss = compare \( y \) and \( \hat{y} = \text{dot}(w,x) \)
- similar but now update \( w \) (user weights) and \( x \) (movie weight)
What loss functions are possible?

<table>
<thead>
<tr>
<th>Loss Function</th>
<th>Definition and Derivatives</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_{NZSL}$</td>
<td>$L_{NZSL} = \sum_{(i,j)\in Z} (V_{ij} - [WH]_{ij})^2$</td>
</tr>
</tbody>
</table>

\[
\frac{\partial}{\partial W_{ik}} L_{ij} = -2(V_{ij} - [WH]_{ij})H_{kj}
\]

\[
\frac{\partial}{\partial H_{kj}} L_{ij} = -2(V_{ij} - [WH]_{ij})W_{ik}
\]
What loss functions are possible?

<table>
<thead>
<tr>
<th>Loss Function</th>
<th>Definition and Derivatives</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_{L2}$</td>
<td>$L_{L2} = L_{NZSL} + \lambda \left( |W|^2_F + |H|^2_F \right)$</td>
</tr>
<tr>
<td></td>
<td>$= \sum_{(i,j) \in \mathcal{Z}} \left[ (V_{ij} - [WH]<em>{ij})^2 + \lambda \left( \frac{|W</em>{i*}|^2_F}{N_{i*}} + \frac{|H_{*j}|^2_F}{N_{*j}} \right) \right]$</td>
</tr>
<tr>
<td></td>
<td>$\frac{\partial}{\partial W_{ik}} L_{ij} = -2(V_{ij} - [WH]<em>{ij})H</em>{kj} + 2\lambda \frac{W_{ik}}{N_{i*}}$</td>
</tr>
<tr>
<td></td>
<td>$\frac{\partial}{\partial H_{kj}} L_{ij} = -2(V_{ij} - [WH]<em>{ij})W</em>{ik} + 2\lambda \frac{H_{kj}}{N_{*j}}$</td>
</tr>
</tbody>
</table>
Stochastic Gradient Descent on Netflix Data

Limited memory quasi-Newton

ALS = alternating least squares
Large-Scale Matrix Factorization with Distributed Stochastic Gradient Descent

Rainer Gemulla

talk pilfered from

Peter J. Haas  Yannis Sismanis  Erik Nijkamp
Averaging Techniques

- Iterative SGD, no mixing
- Limited memory quasi-Newton
- Param mixing
- Alternating least squares
- IPM
Matrix factorization as SGD - why does this work? Here’s the key claim:

Require that the loss can be written as

\[ L = \sum_{(i,j) \in Z} l(V_{ij}, W_{i*}, H_{*,j}) \]

\[
\frac{\partial}{\partial W_{i'k}} L_{ij}(W, H) = \begin{cases} 
0 & \text{if } i \neq i' \\
\frac{\partial}{\partial W_{ik}} l(V_{ij}, W_{i*}, H_{*,j}) & \text{otherwise}
\end{cases} 
\]

\[
\frac{\partial}{\partial H_{kj'}} L_{ij}(W, H) = \begin{cases} 
0 & \text{if } j \neq j' \\
\frac{\partial}{\partial H_{kj}} l(V_{ij}, W_{i*}, H_{*,j}) & \text{otherwise}
\end{cases} 
\]
Problem Structure

- SGD steps depend on each other
  \[ \theta_{n+1} = \theta_n - \epsilon_n \hat{L}'(\theta_n) \]

- An SGD step on example \( z \in \mathbb{Z} \ldots \)
  1. Reads \( W_{i_z} \) and \( H_{*j_z} \)
  2. Performs gradient computation \( L'_{ij}(W_{i_z}, H_{*j_z}) \)
  3. Updates \( W_{i_z} \) and \( H_{*j_z} \)

- Not all steps are dependent
Exploitation

- Block and distribute the input matrix $\mathbf{V}$
- High-level approach (Map only)
  1. Pick a “diagonal”
  2. Run SGD on the diagonal (in parallel)
  3. Merge the results
  4. Move on to next “diagonal”

- Steps 1–3 form a cycle

Node 1

Node 2

Node 3
1. Pick a “diagonal”
2. Run SGD on the diagonal (in parallel)
3. Merge the results
4. Move on to next “diagonal”

<table>
<thead>
<tr>
<th>H1</th>
<th>H2</th>
<th>H3</th>
</tr>
</thead>
<tbody>
<tr>
<td>W1</td>
<td>V 11</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W2</td>
<td>V 22</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W3</td>
<td>V 33</td>
<td></td>
</tr>
</tbody>
</table>

Epoch 1

<table>
<thead>
<tr>
<th>H1</th>
<th>H2</th>
<th>H3</th>
</tr>
</thead>
<tbody>
<tr>
<td>W1</td>
<td>V 12</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W2</td>
<td>V 23</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W3</td>
<td>V 31</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>H1</th>
<th>H2</th>
<th>H3</th>
</tr>
</thead>
<tbody>
<tr>
<td>W1</td>
<td>V 13</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W2</td>
<td>V 21</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W3</td>
<td>V 32</td>
<td></td>
</tr>
</tbody>
</table>
Averaging Techniques

- Iterative SGD, no mixing
- Limited memory quasi-Newton
- Param mixing
- Alternating least squares
- IPM
(a) Netflix, NZSL
(b) Netflix, L2, $\lambda = 50$
(c) Netflix, NZL2, $\lambda = 0.05$
Hadoop scalability

(b) Increasing cores (Hadoop, 6.4B entries)

Hadoop process setup time starts to dominate
MF is like logistic regression
Linear regression as MF

Examples

\[ \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \\ \vdots & \vdots \\ x_{n1} & y_n \end{bmatrix} \]

Weight vectors

\[ \begin{bmatrix} a_1 & a_2 & \cdots & a_m \\ b_1 & b_2 & \cdots & b_m \end{bmatrix} \]

Training data

\[ \begin{bmatrix} v_{11} & \cdots \\ \vdots & \vdots \\ \vdots & \vdots \\ v_{nm} \end{bmatrix} \]

\[ y_1 = x \cdot w_1 \]
\[ y_2 = x \cdot w_2 \]
\[ \vdots \]
\[ y_m = x \cdot w_m \]
Logistic regression as MF

Examples

Weight vectors

Training data

\[ \begin{align*}
X = \begin{bmatrix}
x_{11} & x_{12} \\
x_{21} & x_{22} \\
\vdots & \vdots \\
x_{n1} & y_n
\end{bmatrix},
W = \begin{bmatrix}
a_1 & a_2 & \ldots & a_m \\
b_1 & b_2 & \ldots & b_m
\end{bmatrix},
V = \begin{bmatrix}
v_{11} & \ldots \\
\vdots & \vdots \\
v_{ij} & \ldots \\
v_{nm}
\end{bmatrix}.
\end{align*} \]
Vectorizing logistic regression

• Many ML methods can be rewritten using nothing but vector-matrix operations (“vectorizing”)

• Why do this?
  – Simpler (once you understand it well)
  – Faster (given the right infrastructure - e.g., numpy, GPUs, ...)
  – Can simplify optimization (more later)
Vectorized minibatch logistic regression

• Computation we’d like to vectorize:
  – For each \( \mathbf{x} \) in the minibatch, compute

\[
p = \frac{1}{1 + e^{-\mathbf{x} \cdot \mathbf{w}}} = \frac{1}{1 + \exp(-\sum_j x^j w^j)}
\]

• For each feature \( j \): update \( w^j \) using

\[
\frac{\partial}{\partial w^j} \log P(Y = y|X = \mathbf{x}, \mathbf{w}) = (y - p)x^j
\]
Vectorizing logistic regression

• Computation we’d like to parallelize:
  – For each $\mathbf{x}$ in the minibatch $X_{\text{batch}}$, compute

$$p \equiv \frac{1}{1 + e^{-\mathbf{x} \cdot \mathbf{w}}} = \frac{1}{1 + \exp(-\sum_j x_j w_j)}$$

$$X_{\text{batch}} \mathbf{w} = \begin{bmatrix} x_1^1 & \cdots & x_1^J \\ \vdots & \ddots & \vdots \\ x_B^1 & \cdots & x_B^J \end{bmatrix} \begin{bmatrix} w_1^1 \\ \vdots \\ w_J^1 \end{bmatrix} = \begin{bmatrix} \mathbf{w} \cdot x_1 \\ \vdots \\ \mathbf{w} \cdot x_B \end{bmatrix}$$
Vectorizing logistic regression

• Computation we’d like to parallelize:
  – For each $\mathbf{x}$ in the minibatch $X_{batch}$, compute

$$p = \frac{1}{1 + e^{-\mathbf{x} \cdot \mathbf{w}}} = \frac{1}{1 + \exp(-\sum_j x_j w_j)}$$

$$\begin{bmatrix} w \cdot x_1 \\ \vdots \\ w \cdot x_B \end{bmatrix} + 1$$

in numpy if $M$ is a matrix
$M+1$ does the “right thing”
so does $M$.exp(), $M$.dot(), $M$.reciprocal(), ...
Vectorizing logistic regression

• Computation we’d like to parallelize:
  – For each $x$ in the minibatch, compute

\[
p \equiv \frac{1}{1 + e^{-x \cdot w}} = \frac{1}{1 + \exp(-\sum_j x^j w^j)}
\]

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

def logistic(X): return (X.exp()+1).reciprocal()
p = logistic(Xb.dot(w))  # B rows, 1 column
Binary to softmax logistic regression

\[ p = \frac{1}{1 + e^{-x \cdot w}} = \frac{1}{1 + \exp(-\sum_{j} x^j w^j)} \]

\[
X_{batch} w = \begin{bmatrix}
    x^1_1 & \cdots & x^1_j \\
    \vdots & \ddots & \vdots \\
    x^1_B & \cdots & x^J_B
\end{bmatrix}
\begin{bmatrix}
    w^1 \\
    \vdots \\
    w^J
\end{bmatrix} = 
\begin{bmatrix}
    w \cdot x^1_1 \\
    \vdots \\
    w \cdot x^1_B
\end{bmatrix}
\]
Binary to **softmax** logistic regression

\[
p \equiv \frac{1}{1 + e^{x \cdot w}} = \frac{1}{1 + \exp(-\sum_{j} x^j w^j)}
\]

\[
p^y \equiv \frac{\exp(x \cdot w^y)}{\sum_{y'} \exp(x \cdot w^{y'})}
\]

\[
XW = \begin{bmatrix}
  x_1^1 & \cdots & x_1^J \\
  \vdots & \ddots & \vdots \\
  x_B^1 & \cdots & x_B^J
\end{bmatrix}
\begin{bmatrix}
  w_1^1 \\
  \vdots \\
  w_J^1
\end{bmatrix} = \begin{bmatrix}
  w \cdot x_1 \\
  \vdots \\
  w \cdot x_B
\end{bmatrix}
\]

\[
XW = \begin{bmatrix}
  x_1^1 & \cdots & x_1^J \\
  \vdots & \ddots & \vdots \\
  x_B^1 & \cdots & x_B^J
\end{bmatrix}
\begin{bmatrix}
  w_1^{y1} & \cdots & w_1^{yK} \\
  \vdots & \ddots & \vdots \\
  w_J^{y1} & \cdots & w_J^{yK}
\end{bmatrix} = \begin{bmatrix}
  w^{y1} \cdot x_1 & \cdots & w^{yK} \cdot x_1 \\
  \vdots & \ddots & \vdots \\
  w^{y1} \cdot x_B & \cdots & w^{yK} \cdot x_B
\end{bmatrix}
\]
prob will have B rows and K columns, and each row will sum to 1

\[ p^y \equiv \frac{\exp(x \cdot w^y)}{\sum_{y', \exp(x \cdot w^{y'})}} \]

\[ XW = \begin{bmatrix} x_1^1 & \cdots & x_J^1 \\ \vdots & \ddots & \vdots \\ x_1^B & \cdots & x_J^B \end{bmatrix} \begin{bmatrix} w_1^{y_1} & \cdots & w_1^{y_K} \\ \vdots & \ddots & \vdots \\ w_J^{y_1} & \cdots & w_J^{y_K} \end{bmatrix} = \begin{bmatrix} w_1^{y_1} \cdot x_1^1 & \cdots & w_1^{y_K} \cdot x_1^1 \\ \vdots & \ddots & \vdots \\ w_J^{y_1} \cdot x_B^1 & \cdots & w_J^{y_K} \cdot x_B^1 \end{bmatrix} \]

Matrix multiply, then exponentiate component-wise

Sum the columns to get the denominator; keepdim=True means...

... that this line will work correctly even though 'a' and 'a_sum' have different shapes
```python
import numpy as np
import numpy.random as random
from examples.utils.data_utils import gaussian_cluster_generator as make_data

def predict(w, x):
    a = np.exp(np.dot(x, w))
    a_sum = np.sum(a, axis=1, keepdims=True)
    prob = a / a_sum
    return prob

# Using gradient descent to fit the correct classes.

def train(w, x, loops):
    for i in range(loops):
        prob = predict(w, x)
        loss = -np.sum(label * np.log(prob)) / num_samples
        if i % 10 == 0:
            print('Iter {}, training loss {}'.format(i, loss))
        # gradient descent
        dy = prob - label
        dw = np.dot(data.T, dy) / num_samples
        # update parameters; fixed learning rate of 0.1
        w -= 0.1 * dw

# Initialize training data.
num_samples = 10000
num_features = 500
num_classes = 5
data, label = make_data(num_samples, num_features, num_classes)

# Initialize training weight and train
weight = random.randn(num_features, num_classes)
train(weight, data, 100)
```
import numpy as np
import numpy.random as random
from examples.utils.data_utils import gaussian_cluster_generator as make_data

# Predict the class using multinomial logistic regression (softmax regression).
def predict(w, x):
    a = np.exp(np.dot(x, w))
    a_sum = np.sum(a, axis=1, keepdims=True)
    prob = a / a_sum
    return prob

# Using gradient descent to fit the correct classes.
def train(w, x, loops):
    for i in range(loops):
        prob = predict(w, x)
        loss = -np.sum(label * np.log(prob)) / num_samples
        if i % 10 == 0:
            print('Iter {}, training loss {}'.format(i, loss))
        # gradient descent
        dy = prob - label
        dw = np.dot(data.T, dy) / num_samples
        # update parameters; fixed learning rate of 0.1
        w -= 0.1 * dw

# Initialize training data.
num_samples = 10000
num_features = 500
num_classes = 5
data, label = make_data(num_samples, num_features, num_classes)

# Initialize training weight and train
weight = random.randn(num_features, num_classes)
train(weight, data, 100)
import numpy as np
import numpy.random as random
from examples.utils.data_utils import import gaussian.cluster

# Predict the class using multinomial logistic regression

def predict(x, w):
    return np.argmax(np.dot(data.T, dy) / num_samples)

# gradient descent
dw = np.dot(data.T, dy) / num_samples

# update parameters; fixed learning rate
w -= 0.1 * dw

# Initialize training data.

# Using gradient descent to get the correct best

x.T dy =

\[
\begin{bmatrix}
    x_1^1 & \cdots & x_B^1 \\
    \vdots & \ddots & \vdots \\
    x_1^J & \cdots & x_B^J \\
\end{bmatrix}
\begin{bmatrix}
    dy_{x1}^y1 & \cdots & dy_{x1}^yK \\
    \vdots & \ddots & \vdots \\
    dy_{xB}^y1 & \cdots & dy_{xB}^yK \\
\end{bmatrix}
\]

python bug: should be x.T (transpose)

Error on each example x in batch and each class y

The gradient step!
Vectorizing logistic regression

• Many ML methods can be rewritten using nothing but vector-matrix operations (“vectorizing”)

• Why do this?
  – Simpler (once you understand it well)
  – Faster (given the right infrastructure - e.g., numpy, GPUs, ...)
  – Can simplify optimization (more later)