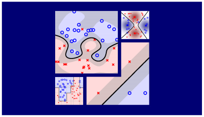
Machine Learning Techniques
(機器學習技法)



Lecture 15: Matrix Factorization

Hsuan-Tien Lin (林軒田)

htlin@csie.ntu.edu.tw

Department of Computer Science
& Information Engineering

National Taiwan University
(國立台灣大學資訊工程系)

Roadmap

1. Embedding Numerous Features: Kernel Models
2. Combining Predictive Features: Aggregation Models
3. Distilling Implicit Features: Extraction Models

Lecture 14: Radial Basis Function Network

linear aggregation of **distance-based similarities** using ***k*-Means clustering** for **prototype finding**

Lecture 15: Matrix Factorization

- Linear Network Hypothesis
- Basic Matrix Factorization
- Stochastic Gradient Descent
- Summary of Extraction Models

Recommender System Revisited

data $\longrightarrow$ $\boxed{\text{ML}}$ $\longrightarrow$ skill

- data: how 'many users' have rated 'some movies'
- skill: predict how a user would rate an unrated movie

A Hot Problem

- competition held by Netflix in 2006
  - 100,480,507 ratings that 480,189 users gave to 17,770 movies
  - 10% improvement = **1 million dollar prize**
- data $\mathcal{D}_m$ for *m*-th movie:

$$\{(\tilde{\mathbf{x}}_n = (n), y_n = r_{nm}) : \text{ user } n \text{ rated movie } m\}$$

—abstract feature $\tilde{\mathbf{x}}_n = (n)$

how to **learn our preferences** from data?

Binary Vector Encoding of Categorical Feature

$\tilde{\mathbf{x}}_n = (n)$: user IDs, such as 1126, 5566, 6211, ...
—called **categorical** features

- **categorical** features, e.g.
  - IDs
  - blood type: A, B, AB, O
  - programming languages: C, C++, Java, Python, ...
- many ML models operate on **numerical** features
  - **linear** models
  - **extended linear** models such as NNet

  —except for **decision trees**
- need: **encoding** (**transform**) from categorical to numerical

binary vector encoding:

$$\mathrm{A} = [1\ 0\ 0\ 0]^T, \mathrm{B} = [0\ 1\ 0\ 0]^T,$$
$$\mathrm{AB} = [0\ 0\ 1\ 0]^T, \mathrm{O} = [0\ 0\ 0\ 1]^T$$

Feature Extraction from Encoded Vector

encoded data $\mathcal{D}_m$ for m-th movie:

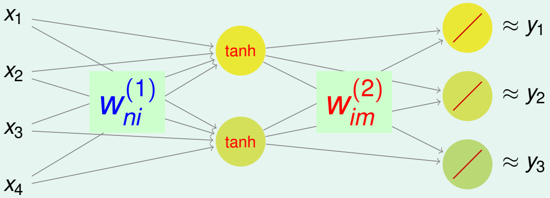
$$\left\{ (\mathbf{x}_n = \text{BinaryVectorEncoding}(n), y_n = r_{nm}) : \text{ user } n \text{ rated movie } m \right\}$$

or, joint data $\mathcal{D}$

$$\left\{ (\mathbf{x}_n = \text{BinaryVectorEncoding}(n), \mathbf{y}_n = [r_{n1} \;\; ? \;\; ? \;\; r_{n4} \;\; r_{n5} \;\; \ldots \;\; r_{nM}]^T) \right\}$$

idea: try **feature extraction** using N-$\tilde{d}$-M NNet without all $x_0^{(\ell)}$

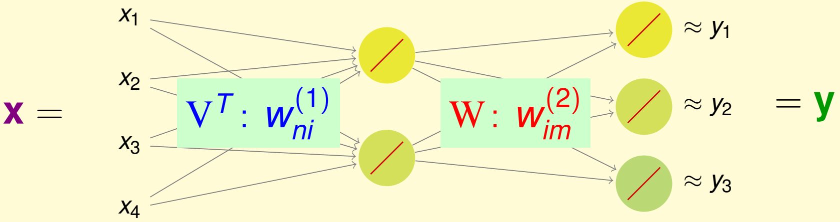
$\mathbf{x} =$



$= \mathbf{y}$

is tanh **necessary**? **:-)**

‘Linear Network’ Hypothesis



$$\left\{\left(\mathbf{x}_n = \text{BinaryVectorEncoding}(n), \mathbf{y}_n = [r_{n1} \ \ ? \ \ ? \ \ r_{n4} \ \ r_{n5} \ \ \ldots \ \ r_{nM}]^T\right)\right\}$$

- rename: V^T for $\left[w_{ni}^{(1)}\right]$ and W for $\left[w_{im}^{(2)}\right]$
- hypothesis: $\mathbf{h}(\mathbf{x}) = \mathrm{W}^T\mathrm{V}\mathbf{x}$
- per-user output: $\mathbf{h}(\mathbf{x}_n) = \mathrm{W}^T\mathbf{v}_n$, where $\mathbf{v}_n$ is n-th column of V

linear network for recommender system:
learn V and W

Fun Time

For N users, M movies, and $\tilde{d}$ 'features', how many variables need to be used to specify a linear network hypothesis $\mathbf{h}(\mathbf{x}) = \mathrm{W}^T\mathrm{V}\mathbf{x}$?

1. $N + M + \tilde{d}$
2. $N \cdot M \cdot \tilde{d}$
3. $(N + M) \cdot \tilde{d}$
4. $(N \cdot M) + \tilde{d}$

Fun Time

For N users, M movies, and $\tilde{d}$ 'features', how many variables need to be used to specify a linear network hypothesis $\mathbf{h}(\mathbf{x}) = \mathrm{W}^T\mathrm{V}\mathbf{x}$?

1. $N + M + \tilde{d}$
2. $N \cdot M \cdot \tilde{d}$
3. $(N + M) \cdot \tilde{d}$
4. $(N \cdot M) + \tilde{d}$

Reference Answer: 3

simply $N \cdot \tilde{d}$ for V^T and $\tilde{d} \cdot M$ for W

Linear Network: Linear Model Per Movie

linear network:

$$\mathbf{h}(\mathbf{x}) = \mathrm{W}^T \underbrace{\mathrm{V}\mathbf{x}}_{\mathbf{\Phi}(\mathbf{x})}$$

—for m-th movie, just linear model $h_m(\mathbf{x}) = \mathbf{w}_m^T \mathbf{\Phi}(\mathbf{x})$
subject to shared transform $\mathbf{\Phi}$

- for every $\mathcal{D}_m$, want $r_{nm} = y_n \approx \mathbf{w}_m^T \mathbf{v}_n$
- E_{in} over all $\mathcal{D}_m$ with squared error measure:

$$E_{\text{in}}(\{\mathbf{w}_m\}, \{\mathbf{v}_n\}) = \frac{1}{\sum_{m=1}^{M} |\mathcal{D}_m|} \sum_{\text{user } n \text{ rated movie } m} \left(r_{nm} - \mathbf{w}_m^T \mathbf{v}_n \right)^2$$

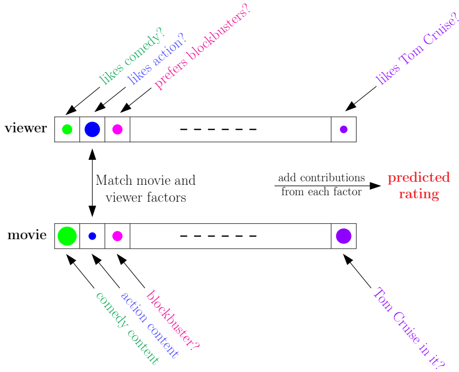
linear network: transform and linear model**S**
jointly learned from all $\mathcal{D}_m$

Matrix Factorization

$$r_{nm} \approx \mathbf{w}_m^T \mathbf{v}_n = \mathbf{v}_n^T \mathbf{w}_m \Longleftrightarrow \mathrm{R} \approx \mathrm{V}^T \mathrm{W}$$

R	movie$_1$	movie$_2$	$\cdots$	movie$_M$
user$_1$	100	80	$\cdots$	?
user$_2$	?	70	$\cdots$	90
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
user$_N$	?	60	$\cdots$	0

$$\approx \begin{array}{c} \mathrm{V}^T \\ \hline \text{—}\mathbf{v}_1^T\text{—} \\ \text{—}\mathbf{v}_2^T\text{—} \\ \cdots \\ \text{—}\mathbf{v}_N^T\text{—} \end{array} \quad \begin{array}{c|cccc} \mathrm{W} & \mathbf{w}_1 & \mathbf{w}_2 & \cdots & \mathbf{w}_M \end{array}$$



Matrix Factorization Model

learning:

known rating

$\rightarrow$ learned **factors** $\mathbf{v}_n$ and $\mathbf{w}_m$

$\rightarrow$ unknown rating prediction

similar modeling can be used for other **abstract features**

Matrix Factorization Learning

$$\min_{\mathrm{W},\mathrm{V}} E_{\text{in}}(\{\mathbf{w}_m\},\{\mathbf{v}_n\}) \propto \sum_{\text{user } n \text{ rated movie } m} \left(r_{nm} - \mathbf{w}_m^T\mathbf{v}_n\right)^2$$
$$= \sum_{m=1}^{M} \left(\sum_{(\mathbf{x}_n, r_{nm}) \in \mathcal{D}_m} \left(r_{nm} - \mathbf{w}_m^T\mathbf{v}_n\right)^2 \right)$$

- two sets of variables:
  can consider **alternating minimization, remember? :-)**
- when $\mathbf{v}_n$ fixed, minimizing $\mathbf{w}_m \equiv$ minimize E_{in} within $\mathcal{D}_m$
  —simply per-movie (per-$\mathcal{D}_m$) **linear regression** without w_0
- when $\mathbf{w}_m$ fixed, minimizing $\mathbf{v}_n$?
  —per-user linear regression without v_0

by **symmetry** between users/movies

called **alternating least squares** algorithm

Alternating Least Squares

Alternating Least Squares

1. initialize $\tilde{d}$ dimension vectors $\{\mathbf{w}_m\}, \{\mathbf{v}_n\}$
2. **alternating optimization** of E_{in}: repeatedly
   1. optimize $\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_M$:
      update $\mathbf{w}_m$ by *m*-th-movie linear regression on $\{(\mathbf{v}_n, r_{nm})\}$
   2. optimize $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_N$:
      update $\mathbf{v}_n$ by *n*-th-user linear regression on $\{(\mathbf{w}_m, r_{nm})\}$

   until **converge**

- **initialize**: usually just **randomly**
- **converge**:
  guaranteed as E_{in} **decreases** during alternating minimization

alternating least squares:
the '**tango**' dance between users/movies

Linear Autoencoder versus Matrix Factorization

Linear Autoencoder

$$\mathrm{X} \approx \mathrm{W}\left(\mathrm{W}^T\mathrm{X}\right)$$

- motivation:
  special d-$\tilde{d}$-d linear NNet
- error measure:
  squared on **all** x_{ni}
- solution: global optimal at
  eigenvectors of $\mathrm{X}^T\mathrm{X}$
- usefulness: extract
  dimension-reduced features

Matrix Factorization

$$\mathrm{R} \approx \mathrm{V}^T\mathrm{W}$$

- motivation:
  N-$\tilde{d}$-M linear NNet
- error measure:
  squared on **known** r_{nm}
- solution: local optimal via
  alternating least squares
- usefulness: extract
  hidden user/movie features

linear autoencoder
$\equiv$ **special** matrix factorization of **complete** X

Fun Time

How many least squares problems does the alternating least squares algorithm needs to solve in one iteration of alternation?

1. number of movies M
2. number of users N
3. $M + N$
4. $M \cdot N$

Fun Time

How many least squares problems does the alternating least squares algorithm needs to solve in one iteration of alternation?

1. number of movies M
2. number of users N
3. $M + N$
4. $M \cdot N$

Reference Answer: 3

simply M per-movie problems and N per-user problems

Another Possibility: Stochastic Gradient Descent

$$E_{\text{in}}(\{\mathbf{w}_m\}, \{\mathbf{v}_n\}) \propto \sum_{\text{user } n \text{ rated movie } m} \underbrace{\left(r_{nm} - \mathbf{w}_m^T \mathbf{v}_n\right)^2}_{\mathrm{err}(\text{user } n,\ \text{movie } m,\ \text{rating } r_{nm})}$$

SGD: randomly pick **one example** within the $\sum$ &
update with **gradient to per-example** err, **remember? :-)**

- **'efficient'** per iteration
- **simple** to implement
- easily extends to **other** err

next: **SGD** for matrix factorization

Gradient of Per-Example Error Function

$$\text{err}(\text{user } n, \text{ movie } m, \text{rating } r_{nm}) = \left(r_{nm} - \mathbf{w}_m^T \mathbf{v}_n\right)^2$$

$$\nabla_{\mathbf{v}_{1126}} \text{err}(\text{user } n, \text{ movie } m, \text{rating } r_{nm}) = \mathbf{0} \text{ unless } n = 1126$$

$$\nabla_{\mathbf{w}_{6211}} \text{err}(\text{user } n, \text{ movie } m, \text{rating } r_{nm}) = \mathbf{0} \text{ unless } m = 6211$$

$$\nabla_{\mathbf{v}_n} \text{err}(\text{user } n, \text{ movie } m, \text{rating } r_{nm}) = -2\left(r_{nm} - \mathbf{w}_m^T \mathbf{v}_n\right)\mathbf{w}_m$$

$$\nabla_{\mathbf{w}_m} \text{err}(\text{user } n, \text{ movie } m, \text{rating } r_{nm}) = -2\left(r_{nm} - \mathbf{w}_m^T \mathbf{v}_n\right)\mathbf{v}_n$$

per-example gradient
$\propto -$(**residual**)(**the other feature vector**)

SGD for Matrix Factorization

SGD for Matrix Factorization

initialize $\tilde{d}$ dimension vectors $\{\mathbf{w}_m\}, \{\mathbf{v}_n\}$ **randomly**

for $t = 0, 1, \ldots, T$

1. randomly pick (n, m) within all known r_{nm}
2. calculate residual $\tilde{r}_{nm} = \left(r_{nm} - \mathbf{w}_m^T \mathbf{v}_n\right)$
3. SGD-update:

$$\begin{aligned} \mathbf{v}_n^{new} &\leftarrow \mathbf{v}_n^{old} + \eta \cdot \tilde{r}_{nm} \mathbf{w}_m^{old} \\ \mathbf{w}_m^{new} &\leftarrow \mathbf{w}_m^{old} + \eta \cdot \tilde{r}_{nm} \mathbf{v}_n^{old} \end{aligned}$$

SGD: perhaps **most popular** large-scale matrix factorization algorithm

SGD for Matrix Factorization in Practice

KDDCup 2011 Track 1: World Champion Solution by NTU

- specialty of data (application need):
  per-user training ratings **earlier than** test ratings **in time**
- training/test mismatch: typical **sampling bias, remember? :-)**

- want: **emphasize latter** examples
- **last** T' iterations of SGD: **only those** T' **examples** considered
  —learned $\{\mathbf{w}_m\}, \{\mathbf{v}_n\}$ **favoring** those
- our idea: **time-deterministic** ~~S~~GD that visits **latter** examples last
  —**consistent improvements** of test performance

> if you **understand** the behavior of techniques,
> easier to **modify** for your real-world use

Fun Time

If all $\mathbf{w}_m$ and $\mathbf{v}_n$ are initialized to the $\mathbf{0}$ vector, what will NOT happen in SGD for matrix factorization?

1. all $\mathbf{w}_m$ are always $\mathbf{0}$
2. all $\mathbf{v}_n$ are always $\mathbf{0}$
3. every residual $\tilde{r}_{nm} =$ the original rating r_{nm}
4. E_{in} decreases after each SGD update

Fun Time

If all $\mathbf{w}_m$ and $\mathbf{v}_n$ are initialized to the $\mathbf{0}$ vector, what will NOT happen in SGD for matrix factorization?

1. all $\mathbf{w}_m$ are always $\mathbf{0}$
2. all $\mathbf{v}_n$ are always $\mathbf{0}$
3. every residual $\tilde{r}_{nm}$ = the original rating r_{nm}
4. E_{in} decreases after each SGD update

Reference Answer: 4

The $\mathbf{0}$ feature vectors provides a per-example gradient of $\mathbf{0}$ for every example. So E_{in} cannot be further decreased.

Map of Extraction Models

extraction models: **feature transform Φ** as **hidden variables**
in addition to linear model

Adaptive/Gradient Boosting

hypotheses g_t; weights α_t

Neural Network/ Deep Learning

weights $w_{ij}^{(\ell)}$;
weights $w_{ij}^{(L)}$

RBF Network

RBF centers $\boldsymbol{\mu}_m$;
weights β_m

Matrix Factorization

user features $\mathbf{v}_n$;
movie features $\mathbf{w}_m$

k Nearest Neighbor

$\mathbf{x}_n$-neighbor RBF;
weights y_n

extraction models: a **rich** family

Map of Extraction Techniques

Adaptive/Gradient Boosting

functional gradient descent

Neural Network/ Deep Learning

SGD (backprop)

autoencoder

RBF Network

***k*-means clustering**

Matrix Factorization

SGD

alternating leastSQR

***k* Nearest Neighbor**

lazy learning :-)

extraction techniques: quite **diverse**

Pros and Cons of Extraction Models

Neural Network/ Deep Learning	RBF Network	Matrix Factorization

Pros

- ‘**easy**’:
  reduces **human burden** in designing features
- **powerful**:
  if **enough** hidden variables considered

Cons

- ‘**hard**’:
  non-convex optimization problems in general
- **overfitting**:
  needs proper **regularization/validation**

be **careful** when applying extraction models

Fun Time

Which of the following extraction model extracts Gaussian centers by k-means and aggregate the Gaussians linearly?

1. RBF Network
2. Deep Learning
3. Adaptive Boosting
4. Matrix Factorization

Fun Time

Which of the following extraction model extracts Gaussian centers by k-means and aggregate the Gaussians linearly?

1. RBF Network
2. Deep Learning
3. Adaptive Boosting
4. Matrix Factorization

Reference Answer: 1

Congratulations on being an **expert** in extraction models! **:-)**

Summary

1. Embedding Numerous Features: Kernel Models
2. Combining Predictive Features: Aggregation Models
3. Distilling Implicit Features: Extraction Models

Lecture 15: Matrix Factorization

- Linear Network Hypothesis
  feature extraction from binary vector encoding
- Basic Matrix Factorization
  alternating least squares between user/movie
- Stochastic Gradient Descent
  efficient and easily modified for practical use
- Summary of Extraction Models
  powerful thus need careful use

- **next: closing remarks of techniques**