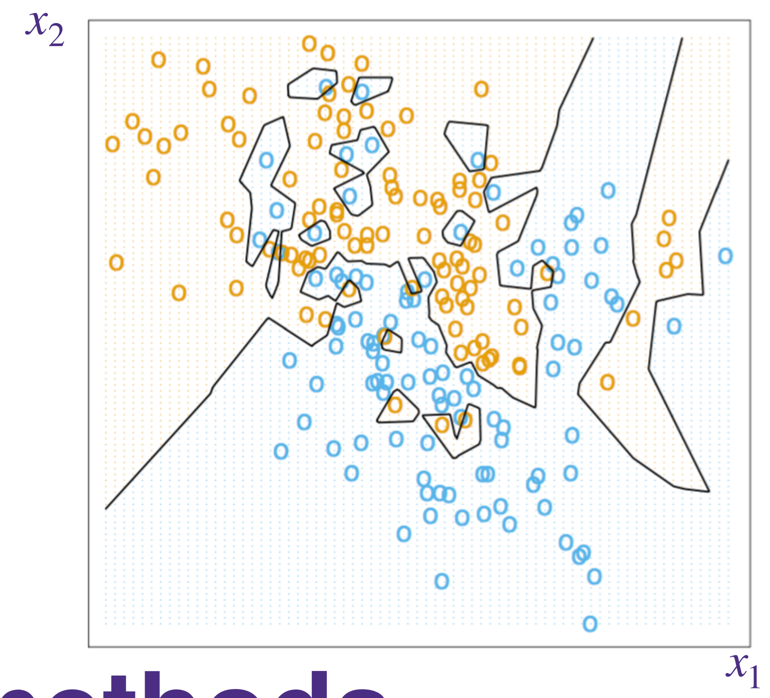
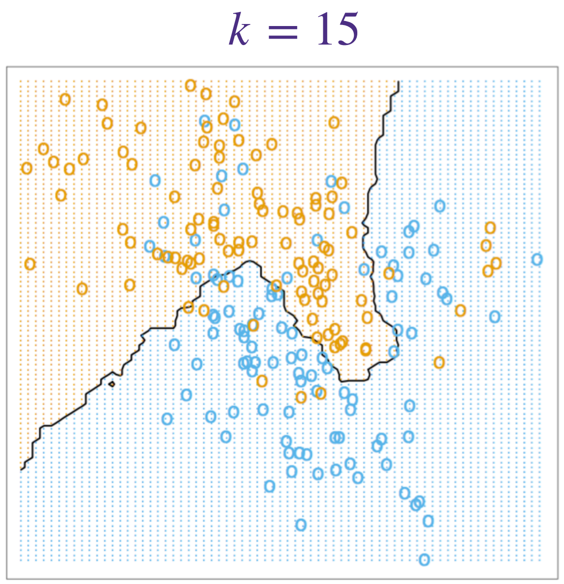
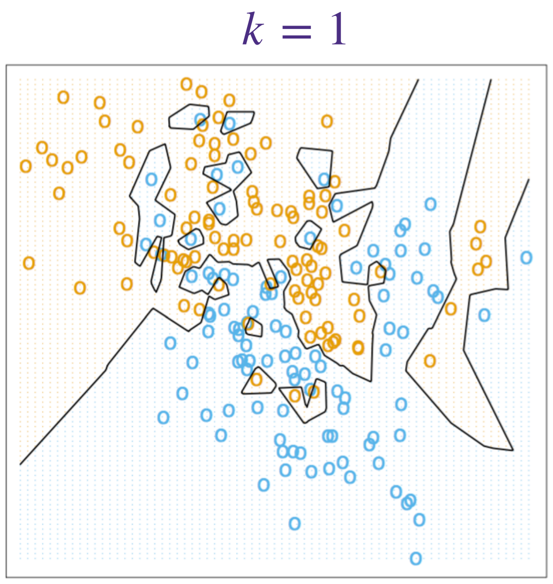
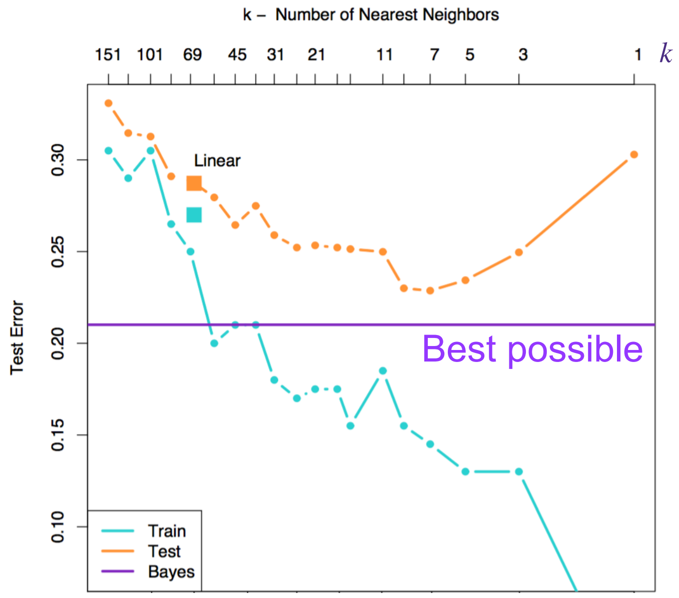


Nearest neighbor methods

Recap of nearest neighbor methods







- **Principle** of designing nearest neighbor methods
  - Consider a "good" estimator that cannot be implemented (because it requires the knowledge of $P_{X,Y}(x, y)$)

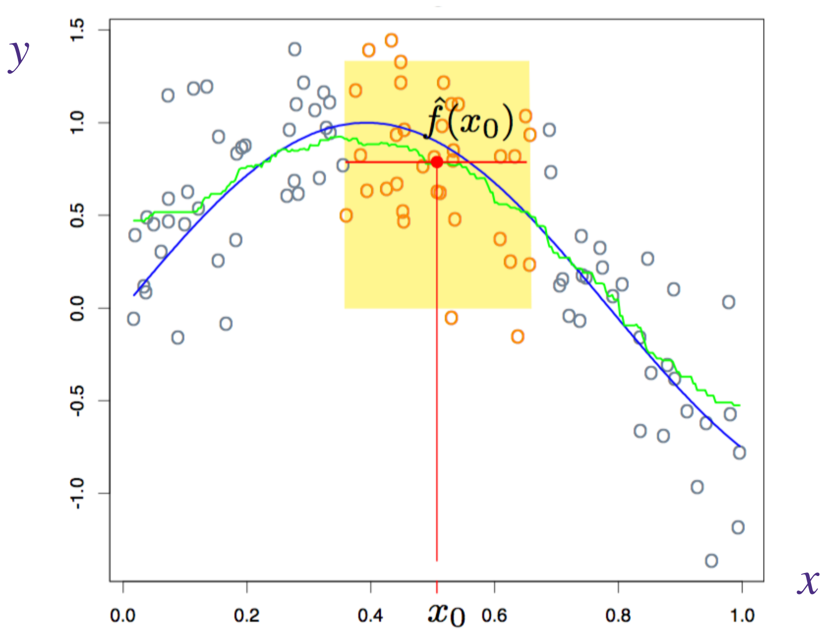
    e.g., for binary classification it is $\hat{y} = +1$ if $P(x, +1) > P(x, -1)$

    -1 if $P(x, +1) < P(x, -1)$
  - Replace $P_{X,Y}(x, y)$ by k_x^y **(i.e.# of nearest neighbors of label** y**)** among k-NNs

    e.g., $\hat{y} = +1$ if $k_x^+ > k_x^-$

    -1 if $k_x^+ < k_x^-$

Consider regression

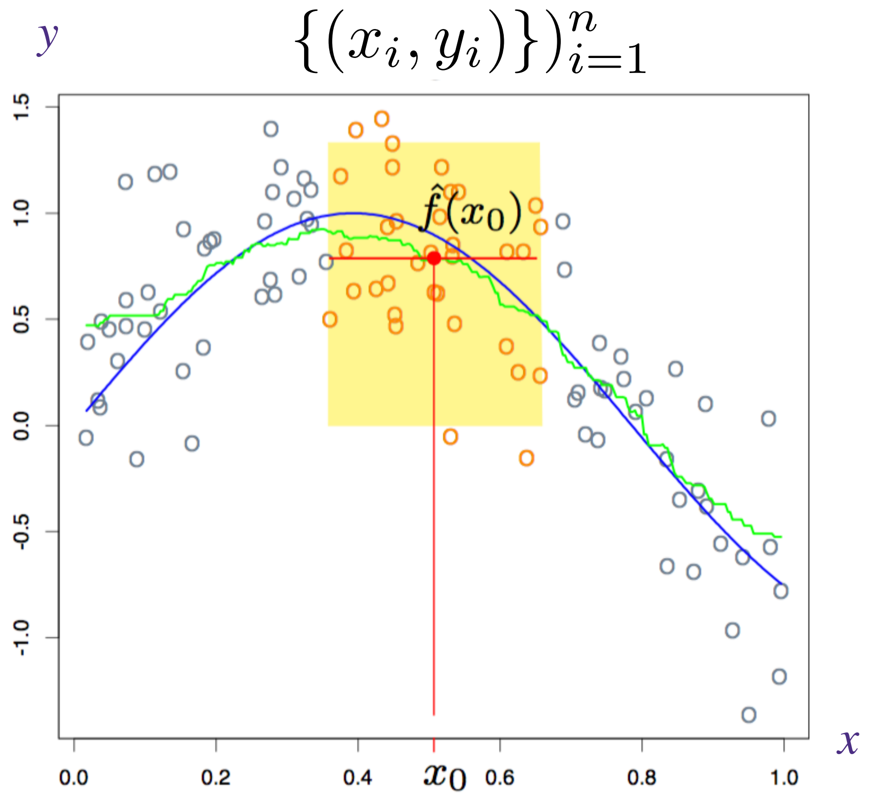


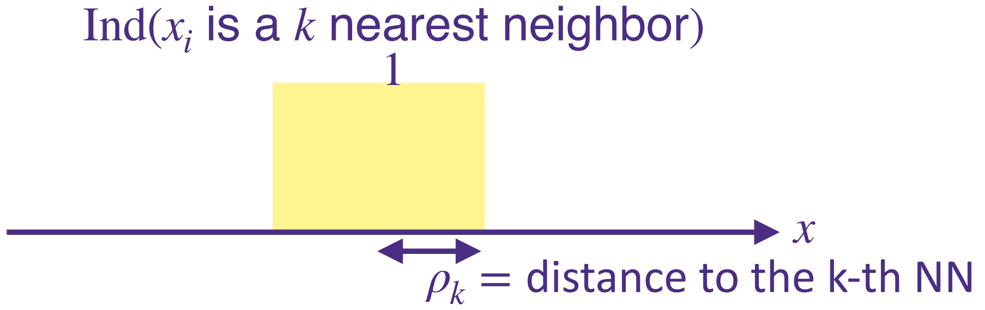
- **Principle** of designing nearest neighbor methods
  - Consider a "good" estimator that cannot be implemented

    e.g., for regression optimal predictor is $\hat{y} = \mathbb{E}[y \,|\, x] = \dfrac{\int y\, P_{X,Y}(x,y)\, dy}{\int 1\, P_{X,Y}(x,y)\, dy}$
  - Replace $P_{X,Y}(x,y)$ by the empirical distribution among k-nearest neighbors

    e.g., $$\hat{y} = \frac{\sum_{j \in \text{nearest neighbor}} y_j}{\sum_{j \in \text{nearest neighbor}} 1} = \frac{\sum_{j \in \text{nearest neighbor}} y_j}{k}$$

Nearest neighbor regression





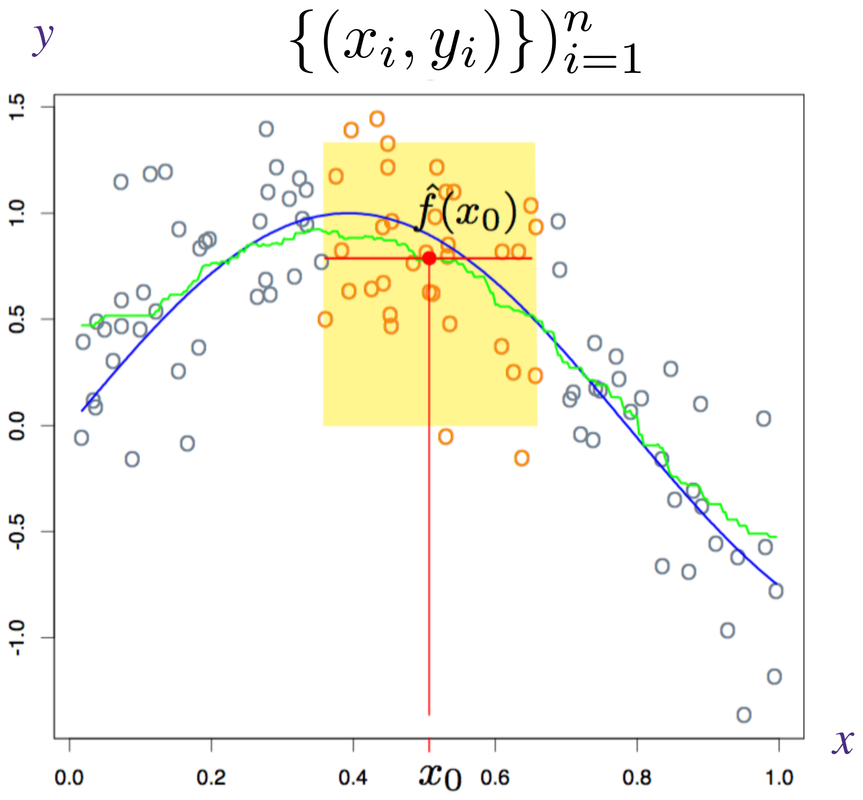
Ind(x_i is a k nearest neighbor)
1
x
ρ_k = distance to the k-th NN

- k-nearest neighbor regressor is

$$\hat{f}(x) = \frac{1}{k} \sum_{j \in \text{nearest neighbor}} y_j$$

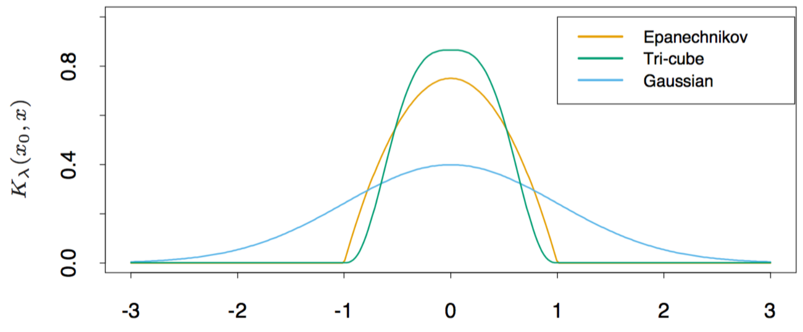
$$= \frac{\sum_{i=1}^n y_i \times \text{Ind}(x_i \text{ is a } k \text{ nearest neighbor})}{\sum_{i=1}^n \text{Ind}(x_i \text{ is a } k \text{ nearest neighbor})}$$

Nearest neighbor regression



Why are far-away neighbors weighted same as close neighbors!

smoothing: $K(x, y)$

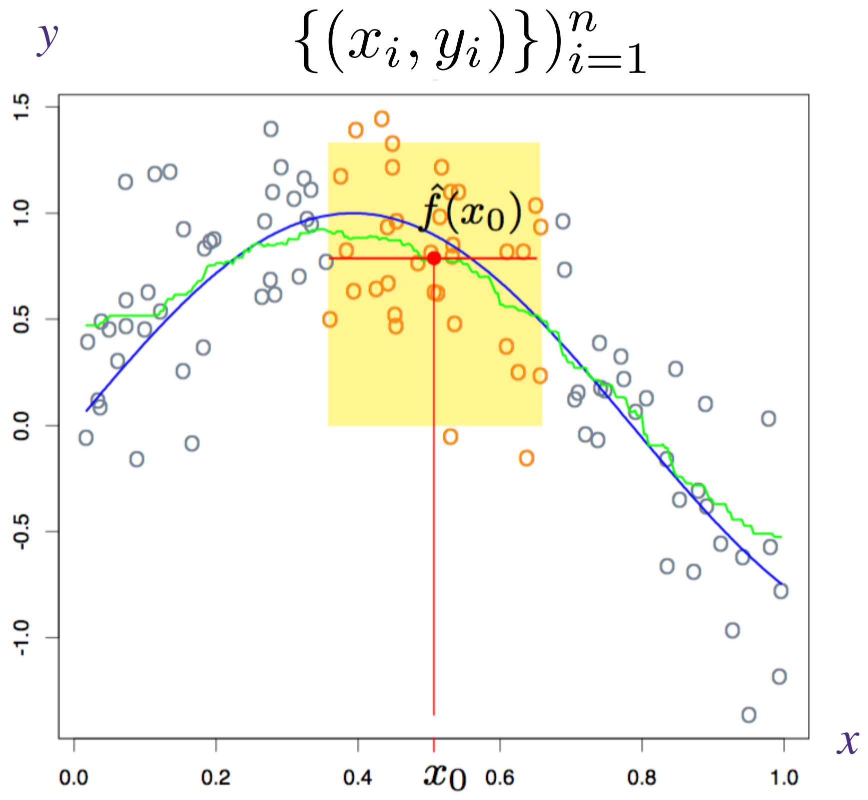


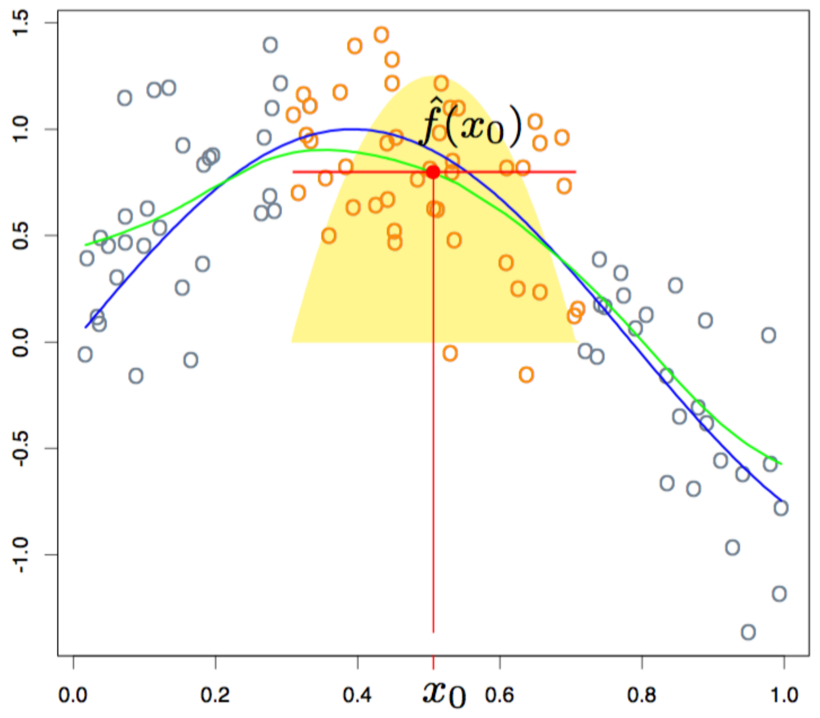
- k-nearest neighbor regressor is

$$\hat{f}(x_0) = \frac{\sum_{i=1}^n y_i \times \text{Ind}(x_i \text{ is a } k \text{ nearest neighbor})}{\sum_{i=1}^n \text{Ind}(x_i \text{ is a } k \text{ nearest neighbor})}$$

$$\widehat{f}(x_0) = \frac{\sum_{i=1}^n K(x_0, x_i) y_i}{\sum_{i=1}^n K(x_0, x_i)}$$

Nearest neighbor regression





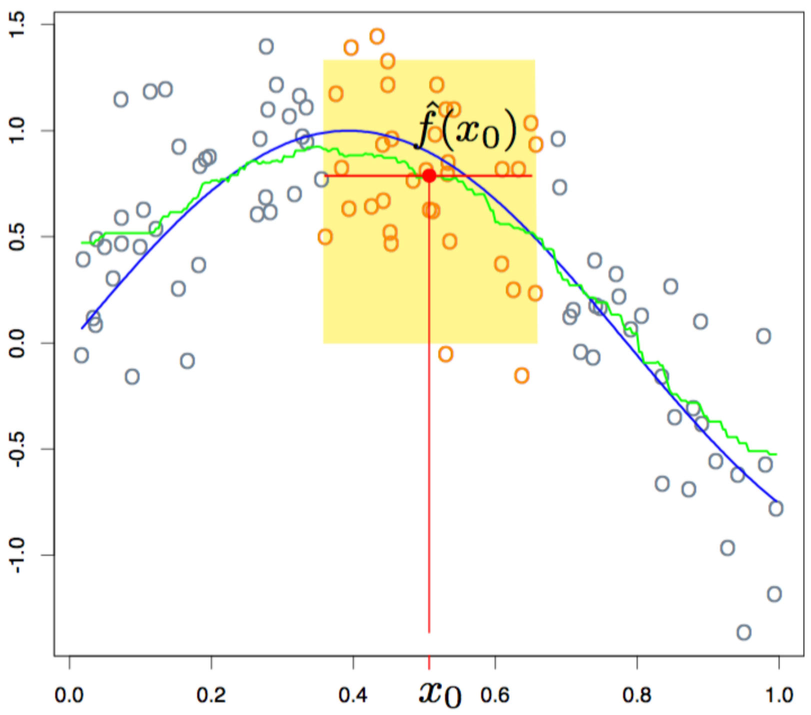
- k-nearest neighbor regressor is

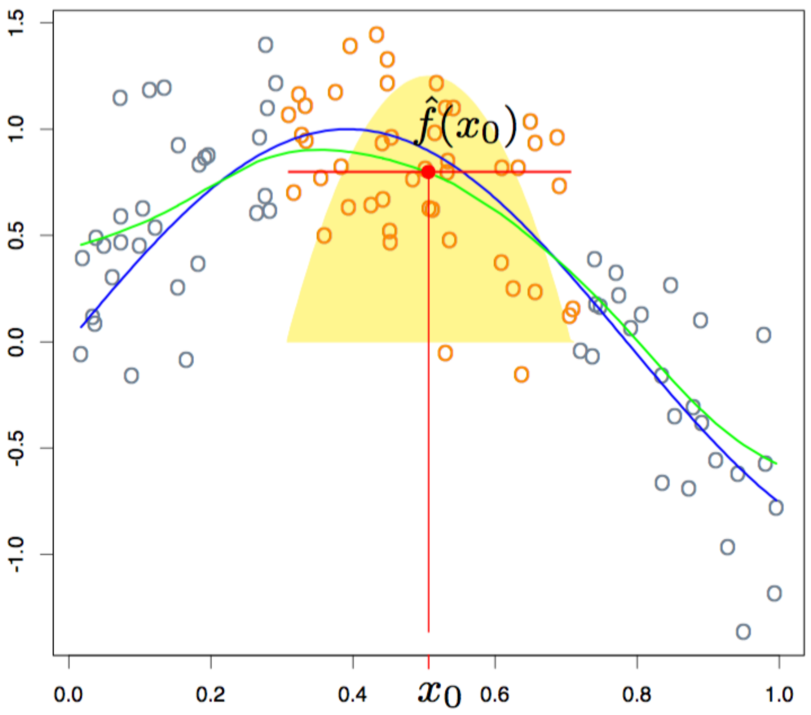
$$\hat{f}(x_0) = \frac{\sum_{i=1}^n y_i \times \text{Ind}(x_i \text{ is a } k \text{ nearest neighbor})}{\sum_{i=1}^n \text{Ind}(x_i \text{ is a } k \text{ nearest neighbor})}$$

$$\widehat{f}(x_0) = \frac{\sum_{i=1}^n K(x_0, x_i) y_i}{\sum_{i=1}^n K(x_0, x_i)}$$

Nearest neighbor regression

$$\{(x_i, y_i)\}_{i=1}^n$$





- k-nearest neighbor regressor is

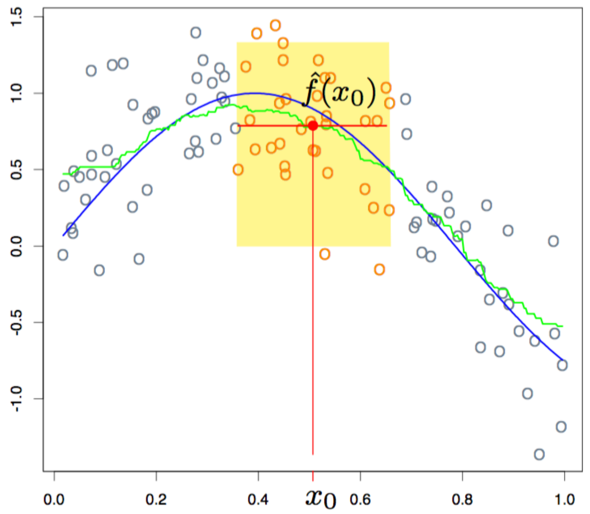
$$\hat{f}(x_0) = \frac{1}{k} \sum_{j \in \text{nearest neighbor}} y_j$$

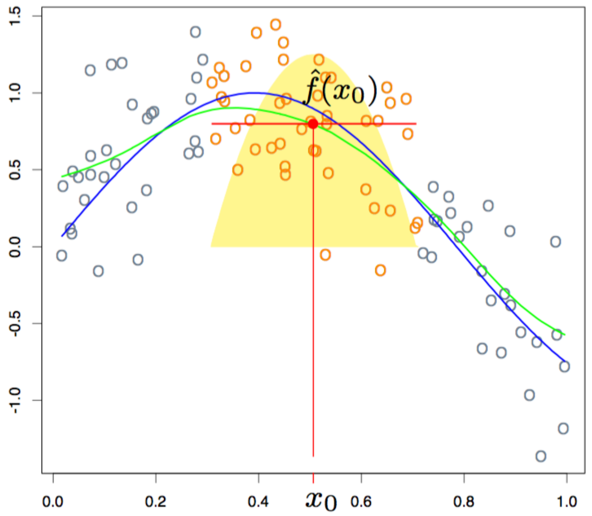
Why just average them?

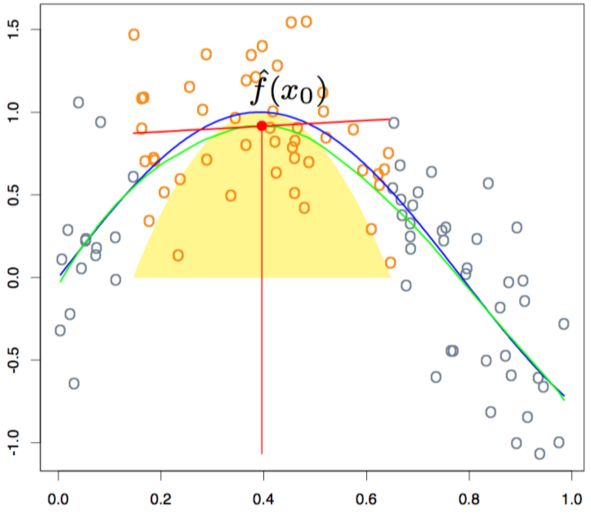
$$\widehat{f}(x_0) = \frac{\sum_{i=1}^n K(x_0, x_i) y_i}{\sum_{i=1}^n K(x_0, x_i)}$$

Nearest neighbor regression

$$\{(x_i, y_i)\}_{i=1}^n$$







$$\hat{f}(x_0) = \frac{\sum_{i=1}^n y_i \times \text{Ind}(x_i \text{ is a } k \text{ nearest neighbor})}{\sum_{i=1}^n \text{Ind}(x_i \text{ is a } k \text{ nearest neighbor})}$$

$$\widehat{f}(x_0) = \frac{\sum_{i=1}^n K(x_0, x_i) y_i}{\sum_{i=1}^n K(x_0, x_i)}$$

$$\widehat{f}(x_0) = b(x_0) + w(x_0)^T x_0$$

$$w(x_0), b(x_0) = \arg\min_{w,b} \sum_{i=1}^n K(x_0, x_i)(y_i - (b + w^T x_i))^2$$

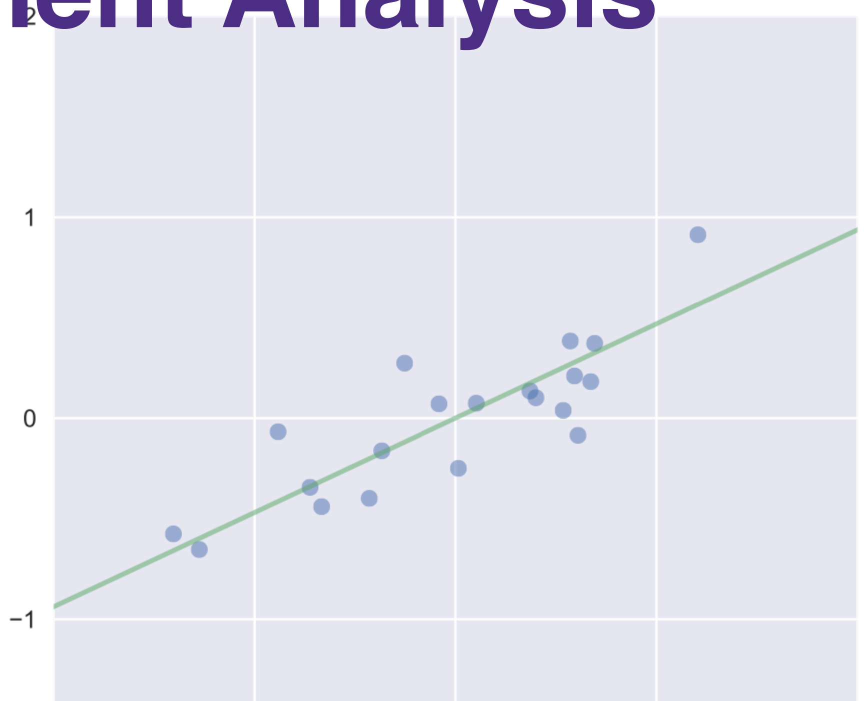
Local Linear Regression

Nearest Neighbor Overview

- Very simple to explain and implement
- No training! But finding nearest neighbors in large dataset at test can be computationally demanding (KD-trees help)
- You can use other forms of distance (not just Euclidean)
- Smoothing and local linear regression can improve performance (at the cost of higher variance)
- With a lot of data, “local methods” have strong, simple theoretical guarantees.
- Without a lot of data, neighborhoods aren’t “local” and methods suffer (curse of dimensionality).

Questions?

Principal Component Analysis



Motivation: dimensionality reduction

- it takes $n \times d$ memory to store data $\{x_i\}_{i=1}^n$ with $x_i \in \mathbb{R}^d$
- but many real data have repeated patterns
- can we represent each image compactly,
  but still preserve most of information, by exploiting similarities?

d pixels per image

n images

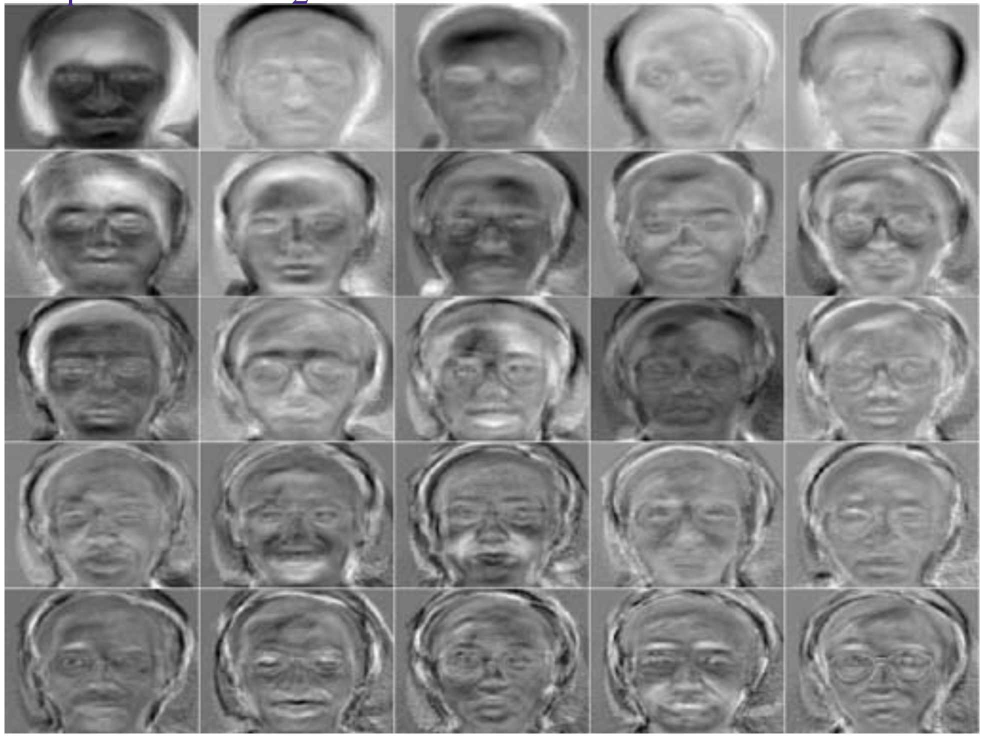
$d \times n$ real values to store the data

Principal component analysis finds a compact linear representation

- patterns that capture the distinct features of the samples is called **principal component** (to be formally defined later)
- we use $r = 25$ principal components

Principal components:

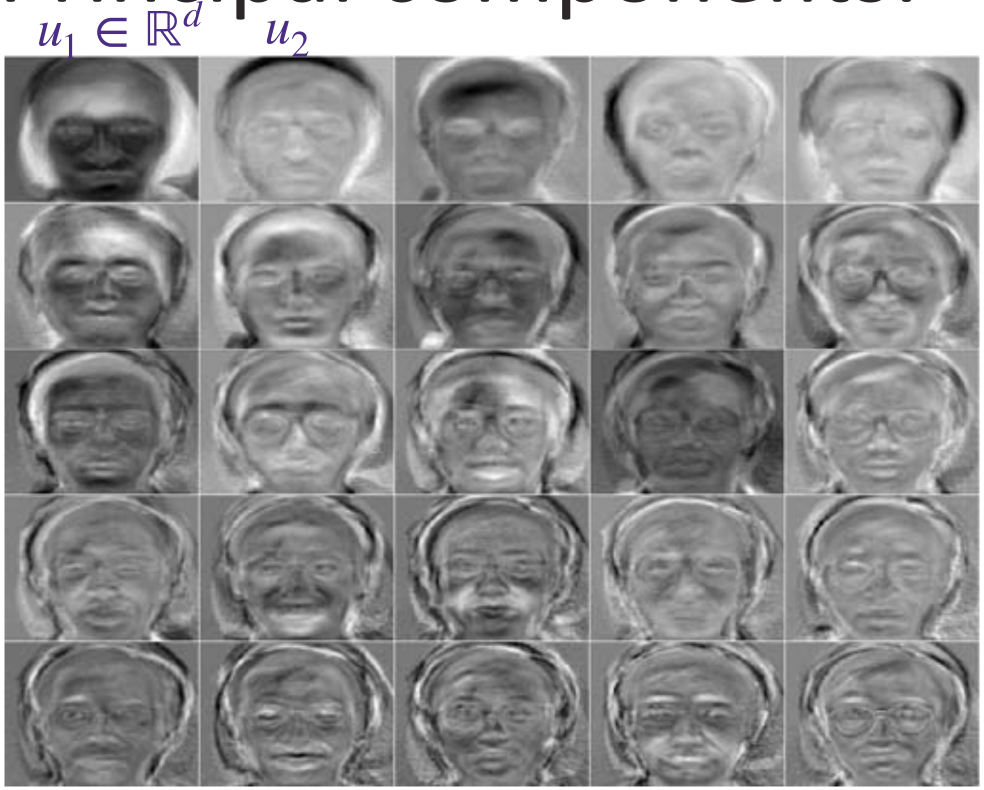
$u_1 \in \mathbb{R}^d$ u_2



Principal component analysis finds a compact linear representation

- patterns that capture the distinct features of the samples is called **principal component** (to be formally defined later)
- we use $r = 25$ principal components
- we can represent each sample as a **weighted linear combination** of the principal components, and just store the weights (as opposed to all pixel values)

Principal components:

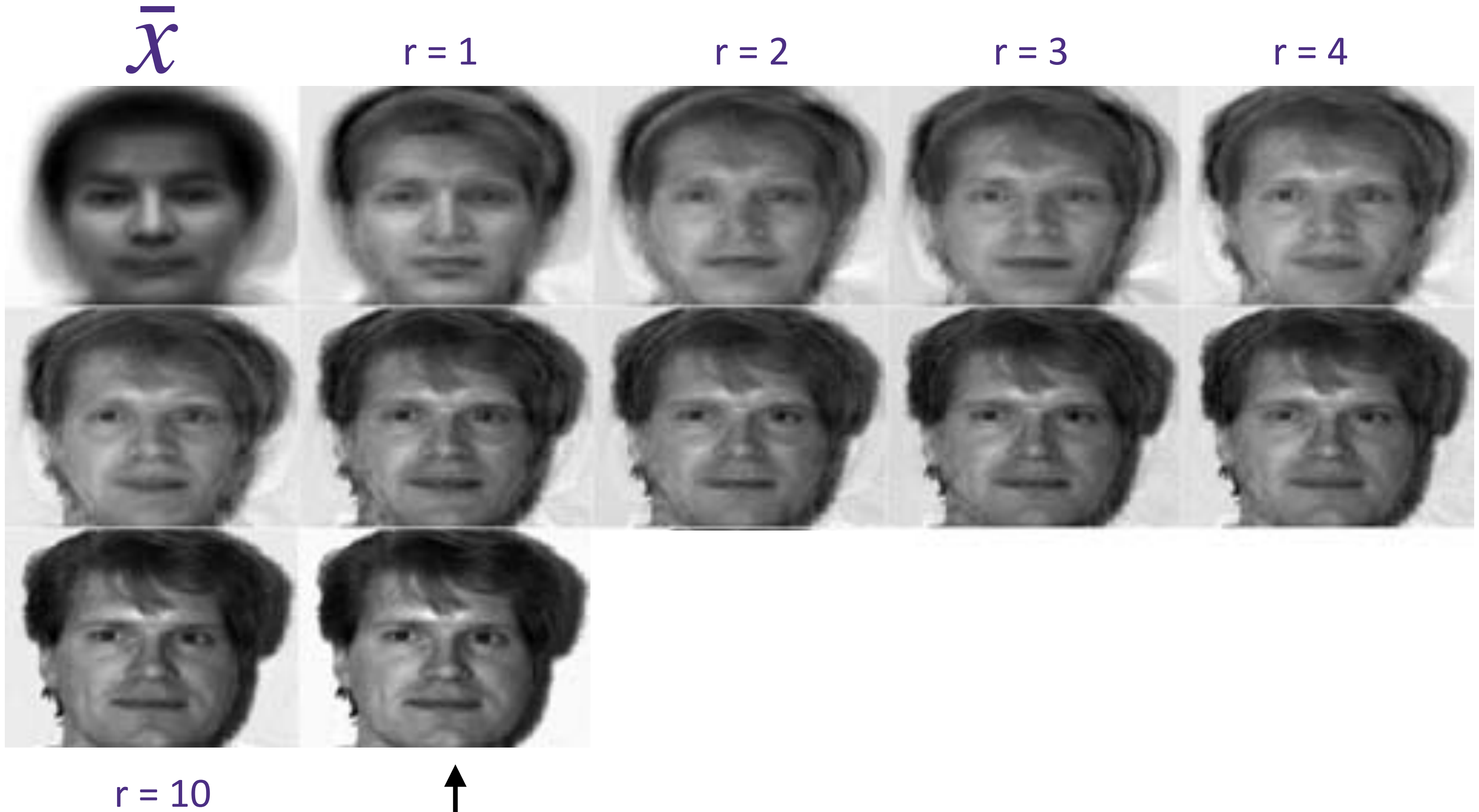


$\approx \; a[1]u_1 + a[2]u_2 + \cdots + a[25]u_{25}$

- Each image is now represented by $r = 25$ numbers $a = (a[1], \ldots, a[25])$
- To store n images, it requires memory of only $d \times r + r \times n \;\ll\; d \times n$

10 principal components give a pretty good reconstruction of a face

average face $\bar{x} + a[1]u_1$ $\bar{x} + a[1]u_1 + a[2]u_2$



Ground truths real face

Assumption

- Notice how we started with the average face $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$
- PCA is applied to $\{x_i - \bar{x}\}_{i=1}^{n}$
- For simplicity, we will assume that x_i's are centered such that $\frac{1}{n}\sum_{i=1}^{n} x_i = 0$
- otherwise, without loss of generality,
  everything we do can be applied to the re-centered version of the data,
  i.e. $\{x_i - \bar{x}\}_{i=1}^{n}$, with $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$

How do we define the principal components?

- Dimensionality reduction (for some $r \ll d$):
  we would like to have a set of orthogonal directions $u_1, \ldots, u_r \in \mathbb{R}^d$, with $\|u_j\|_2 = 1$for all j, such that each data can be represented as linear combination of those direction vectors, i.e.

$$x_i \;\approx\; p_i \;=\; a_i[1]u_1 + \cdots + a_i[r]u_r$$

$$x_i = \begin{bmatrix} x_i[1] \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ x_i[d] \end{bmatrix} \longrightarrow a_i = \begin{bmatrix} a_i[1] \\ \vdots \\ a_i[r] \end{bmatrix}$$

How do we find the principal components?

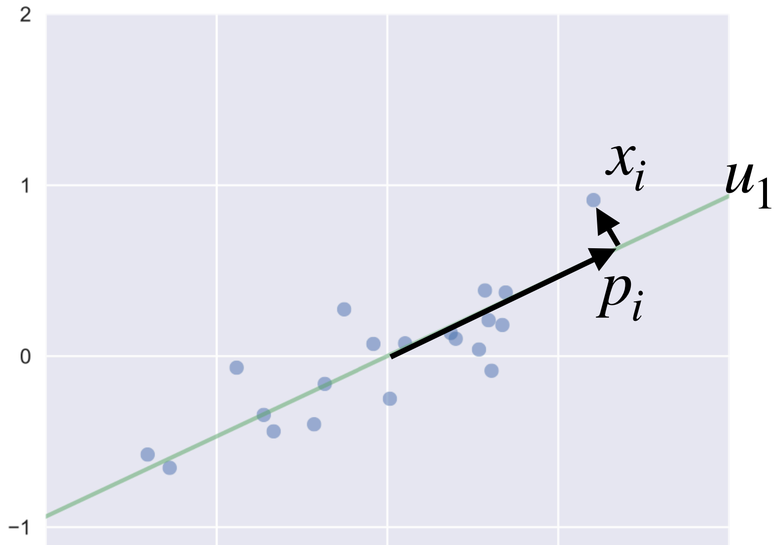
- Dimensionality reduction (for some $r \ll d$):
  we would like to have a set of orthogonal directions $u_1, \ldots, u_r \in \mathbb{R}^d$, with $\|u_j\|_2 = 1$ for all j, such that each data can be represented as linear combination of those direction vectors, i.e.
  $$x_i \;\approx\; p_i \;=\; a_i[1]u_1 + \cdots + a_i[r]u_r$$

$$x_i = \begin{bmatrix} x_i[1] \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ x_i[d] \end{bmatrix} \longrightarrow a_i = \begin{bmatrix} a_i[1] \\ \vdots \\ a_i[r] \end{bmatrix}$$

- those directions that minimize the average reconstruction error for a dataset is called the **principal components**

- given a choice of $u_1, \ldots, u_r$,
  the best representation p_i of x_i
  is the projection of the point onto
  the subspace spanned by u_j's, i.e.

$$a_i[j] = u_j^T x_i$$
$$p_i = \sum_{j=1}^{r} \underbrace{(u_j^T x_i)}_{a_i[j]} u_j$$



- we will use these without proving it

Principal components is the subspace that minimizes the reconstruction error

$$\underset{u_1,\ldots,u_r}{\text{minimize}} \quad \frac{1}{n}\sum_{i=1}^{n} \|x_i - p_i\|_2^2$$

$$p_i = \sum_{j=1}^{r} (u_j^T x_i) u_j = \mathbf{U}\mathbf{U}^T x_i$$

where $\mathbf{U} = [u_1 \quad u_2 \quad \cdots \quad u_r] \in \mathbb{R}^{d\times r}$

$$\boxed{\begin{aligned} &\underset{U}{\text{minimize}} \quad \frac{1}{n}\sum_{i=1}^{n} \|x_i - \mathbf{U}\mathbf{U}^T x_i\|_2^2 \\ &\text{subject to} \quad \mathbf{U}^T\mathbf{U} = \mathbf{I}_{r\times r} \end{aligned}}$$

Q. How do we solve this optimization?

Minimizing reconstruction error to find principal components

$$\underset{U}{\text{minimize}} \ \frac{1}{n}\sum_{i=1}^{n} \|x_i - \mathbf{U}\mathbf{U}^T x_i\|_2^2$$

$$\text{subject to} \ \mathbf{U}^T\mathbf{U} = \mathbf{I}_{r\times r}$$

Minimizing reconstruction error to find principal components

$$\frac{1}{n}\sum_{i=1}^{n}\|x_i - UU^T x_i\|_2^2$$

$$= \frac{1}{n}\sum_{i=1}^{n}\left\{\|x_i\|_2^2 - 2x_i^T UU^T x_i + x_i^T U \underbrace{U^T U}_{=\mathbf{I}} U^T x_i\right\}$$

$$= \underbrace{\frac{1}{n}\sum_{i=1}^{n}\|x_i\|_2^2}_{\text{does not depend on } U} - \frac{1}{n}\sum_{i=1}^{n} x_i^T UU^T x_i$$

$$= C - \sum_{j=1}^{r} \underbrace{\frac{1}{n}\sum_{i=1}^{n}(u_j^T x_i)^2}_{\text{Variance in direction } u_j}$$

$$\underset{U}{\text{minimize}} \quad \frac{1}{n}\sum_{i=1}^{n}\|x_i - \mathbf{U}\mathbf{U}^T x_i\|_2^2$$

$$\text{subject to} \quad \mathbf{U}^T\mathbf{U} = \mathbf{I}_{r\times r}$$

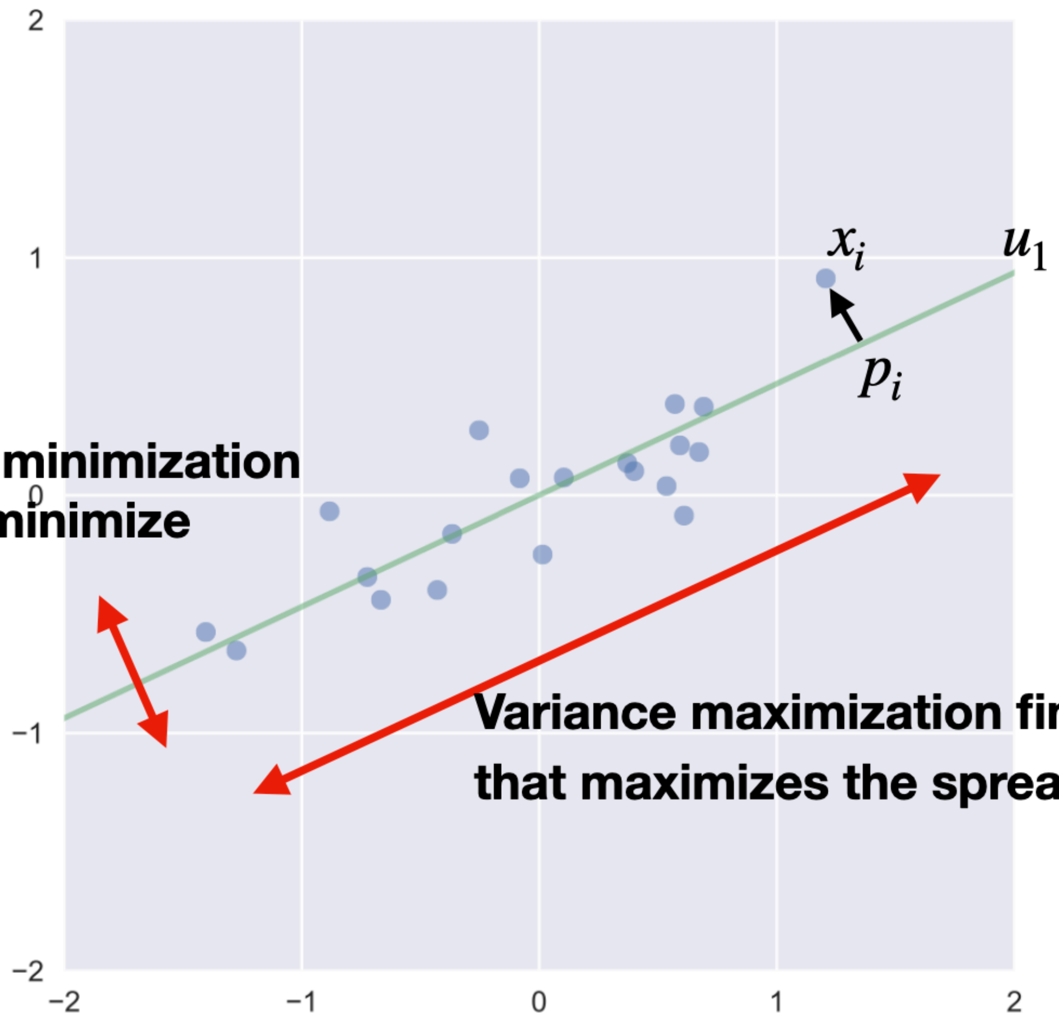
$$\Updownarrow$$

$$\underset{U}{\text{maximize}} \quad \sum_{j=1}^{r}\frac{1}{n}\sum_{i=1}^{n}(u_j^T x_i)^2$$

$$\text{subject to} \quad \mathbf{U}^T\mathbf{U} = \mathbf{I}_{r\times r}$$

Variance maximization vs. reconstruction error minimization

- both give the same principal components as optimal solution



Reconstruction error minimization finds directions that minimize the distances to p_i's

Variance maximization finds directions that maximizes the spread of p_i's

Maximizing variance to find principal components

$$\underset{U}{\text{maximize}} \quad \sum_{j=1}^{r} \frac{1}{n} \sum_{i=1}^{n} (u_j^T x_i)^2$$

$$\text{subject to} \quad \mathbf{U}^T\mathbf{U} = \mathbf{I}_{r\times r}$$

We will solve it for $r = 1$ case,
and the general case follows similarly

$$\underset{u:\|u\|_2=1}{\text{maximize}} \quad \frac{1}{n} \sum_{i=1}^{n} (u^T x_i)^2$$

$$\underset{u:\|u\|_2=1}{\text{maximize}} \quad u^T C u$$

Maximizing variance to find principal components

$$\text{maximize}_u \; u^T \mathbf{C} u \qquad (a)$$

subject to $\|u\|_2^2 = 1$

- we first claim that this optimization problem has the same optimal solution as the following **inequality constrained** problem

$$\text{maximize}_u \; u^T \mathbf{C} u \qquad (b)$$

subject to $\|u\|_2^2 \leq 1$

- the reason is that, because $u^T \mathbf{C} u \geq 0$ for all $u \in \mathbb{R}^d$, the optimal solution of (b) has to have $\|u\|_2^2 = 1$
- if it did not have $\|u\|_2^2 = 1$, say $\|u\|_2^2 = 0.9$, then we can just multiply this u by a constant factor of $\sqrt{10/9}$ and increase the objective by a factor of $10/9$ while still satisfying the constraints

$$\text{maximize}_u \; u^T \mathbf{C} u \qquad (b)$$

$$\textbf{subject to} \quad \|u\|_2^2 \leq 1$$

- we are maximizing the variance, while **keeping u small**
- this can be reformulated as an unconstrained problem, with Lagrangian encoding, to move the constraint into the objective

$$\text{maximize}_u \quad \underbrace{u^T \mathbf{C} u - \lambda \|u\|_2^2}_{F_\lambda(u)} \qquad (c)$$

- this encourages small u as we want, and we can make this connection precise: there exists a (unknown) choice of λ such that the optimal solution of (c) is the same as the optimal solution of (b)
- further, for this choice of λ, the optimal u has $\|u\|_2 = 1$

Solving the unconstrained optimization

$$\text{maximize}_u \quad \underbrace{u^T \mathbf{C} u - \lambda \|u\|_2^2}_{F_\lambda(u)}$$

- to find such λ and the corresponding u, we solve the unconstrained optimization, by setting the gradient to zero
$$\nabla F_\lambda(u) \;=\; 2\mathbf{C}u - 2\lambda u \qquad = \; 0$$
- the candidate solution satisfies: $\mathbf{C}u = \lambda u$, i.e. an eigenvector of $\mathbf{C}$
- let $(\lambda^{(1)}, u^{(1)})$ denote the largest eigenvalue and corresponding eigenvector of $\mathbf{C}$, with norm one, i.e. $\|u^{(1)}\|_2^2 = 1$
- The maximum is. achieved when $u = u^{(1)}$

The principal component analysis

- so far we considered finding ONE principal component $u \in \mathbb{R}^d$
- it is the eigenvector corresponding to the maximum eigenvalue of the covariance matrix

$$\mathbf{C} = \frac{1}{n}\mathbf{X}^T\mathbf{X} \in \mathbb{R}^{d \times d}$$

- We can use Singular Value Decomposition (SVD) to find such eigen vector
- note that is the data is not centered at the origin, we should re-center the data before applying SVD

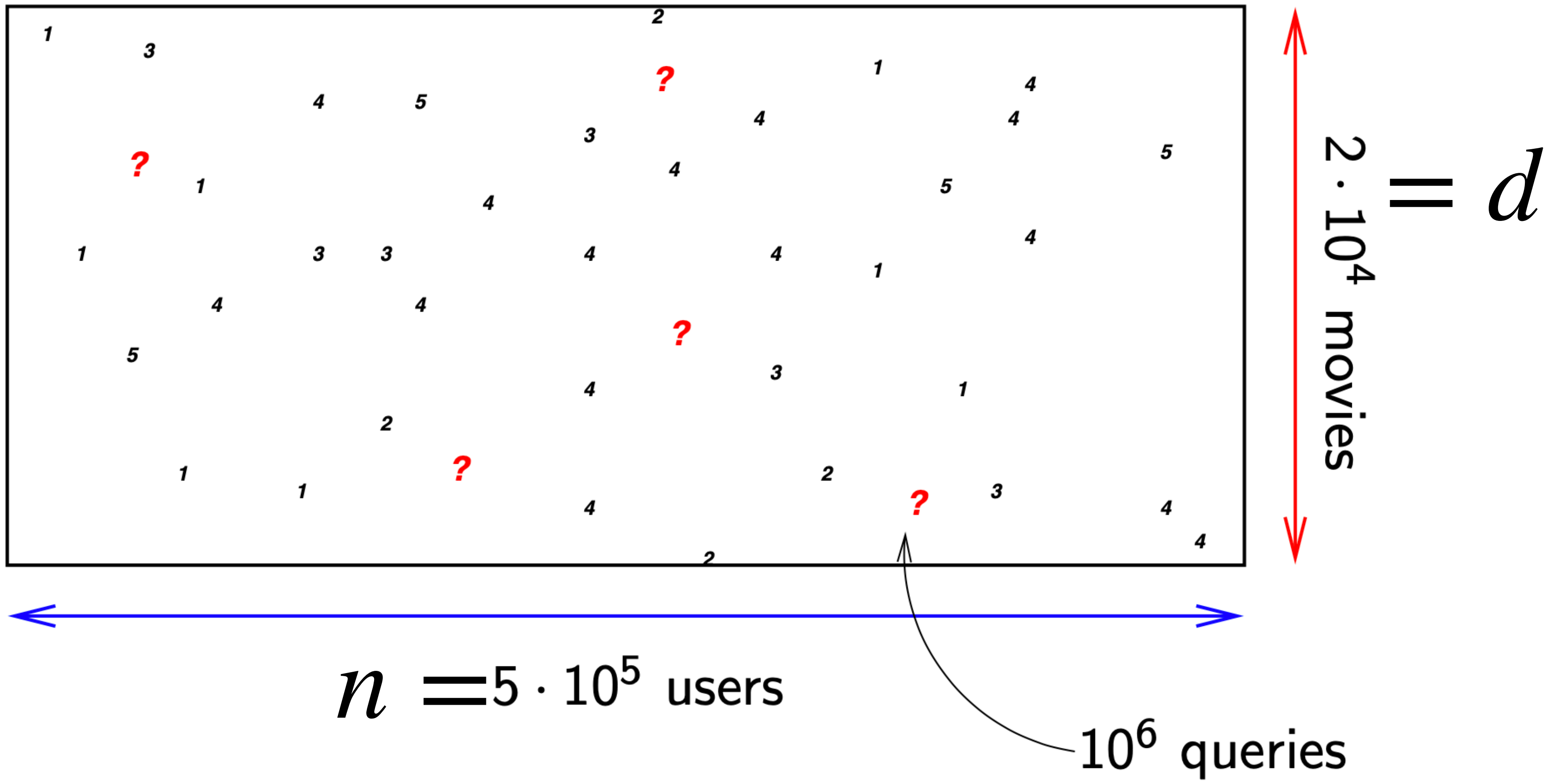
- in general we define and use multiple principal components
- if we need r principal components, we take r eigenvectors corresponding to the largest r eigenvalues of $\mathbf{C}$

Algorithm: Principal Component Analysis

- **input**: data points $\{x_i\}_{i=1}^n$, target dimension $r \ll d$
- **output**: r-dimensional subspace U
- **algorithm**:
  - compute mean $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$
  - compute covariance matrix
    $$\mathbf{C} = \frac{1}{n}\sum_{i=1}^{n} (x_i - \bar{x})(x_i - \bar{x})^T$$
  - let $(u_1, \ldots, u_r)$ be the set of (normalized) eigenvectors with corresponding to the largest r eigenvalues of $\mathbf{C}$
  - return $\mathbf{U} = [u_1 \quad u_2 \quad \cdots \quad u_r]$

- further the data points can be represented compactly via
  $$a_i = \mathbf{U}^T(x_i - \bar{x}) \in \mathbb{R}^r$$

Matrix completion for recommendation systems



- users provide ratings on a few movies, and we want to predict the missing entries in this ratings matrix, so that we can make recommendations
- without any assumptions, the missing entries can be anything, and no prediction is possible

Matrix completion

- however, the ratings are not arbitrary, but people with similar tastes rate similarly
- such structure can be modeled using low dimensional representation of the data as follows
- we will find a set of principal component vectors

$$\mathbf{U} = [u_1 \quad u_2 \quad \cdots \quad u_r] \; \in \mathbb{R}^{d \times r}$$

- such that that ratings $x_i \in \mathbb{R}^d$ of user i, can be represented as

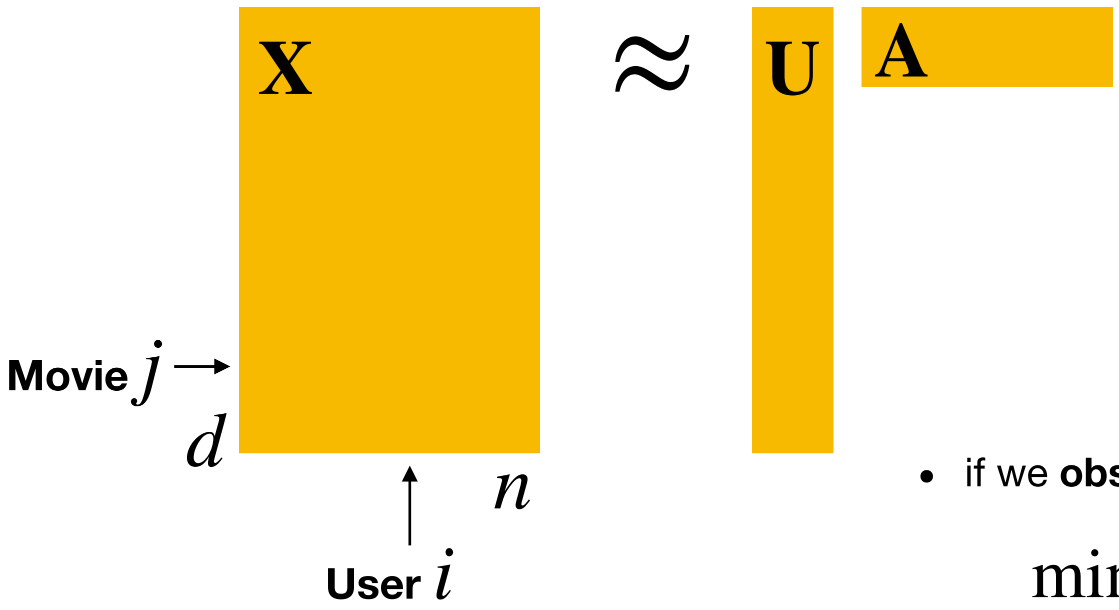
$$\begin{aligned} x_i &= a_i[1]u_1 + \cdots a_i[r]u_r \\ &= \mathbf{U}a_i \end{aligned}$$

  for some lower-dimensional $a_i \in \mathbb{R}^r$ for i-th user and some $r \ll d$

- for example, $u_1 \in \mathbb{R}^d$ means how horror movie fans like each of the d movies,
- and $a_i[1]$ means how much user i is fan of horror movies

Matrix completion

- let $\mathbf{X} = [x_1 \quad x_2 \quad \cdots \quad x_n] \in \mathbb{R}^{d \times n}$ be the ratings matrix, and assume it is fully observed, i.e. we know all the entries
- then we want to find $\mathbf{U} \in \mathbb{R}^{d \times r}$ and $\mathbf{A} = [a_1 \quad a_2 \quad \cdots \quad a_n] \in \mathbb{R}^{r \times n}$ that approximates $\mathbf{X}$



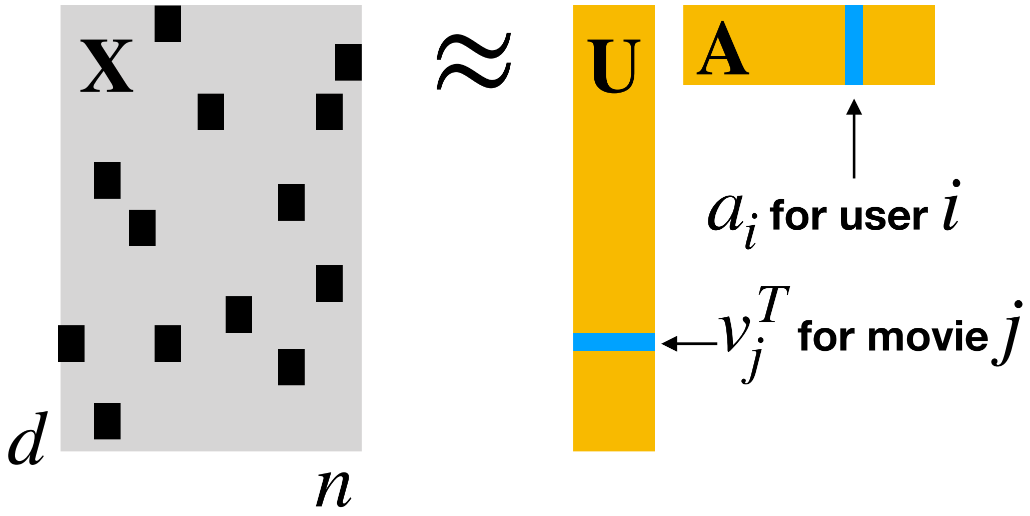
- if we **observe all entries** of $\mathbf{X}$, then we can solve

$$\text{minimize}_{\mathbf{U},\mathbf{A}} \sum_{i=1}^{n} \|x_i - \mathbf{U}a_i\|_2^2$$

which can be solved using PCA (i.e. SVD)

Matrix completion

- in practice, we only observe $\mathbf{X}$ partially
- let $S_{\text{train}} = \{(i_\ell, j_\ell)\}_{\ell=1}^{N}$ denote N observed ratings for user i_ℓ on movie j_ℓ



- let v_j^T denote the j-th row of $\mathbf{U}$ and a_i denote i-th column of $\mathbf{A}$
- then user i's rating on movie j, i.e. $\mathbf{X}_{ji}$ is approximated by $v_j^T a_i$, which is the inner product of v_j (a column vector) and a column vector a_i
- we can also write it as $\langle v_j, a_i \rangle = v_j^T a_i$

Matrix completion

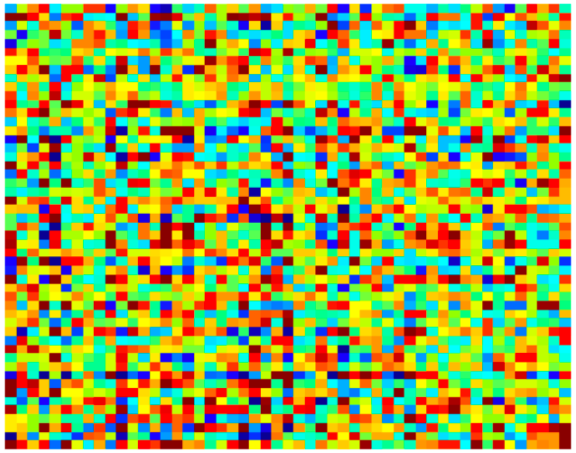
- a natural approach to fit v_j's and $a_i's$ to given training data is to solve

$$\text{minimize}_{\mathbf{U},\mathbf{A}} \sum_{(i,j)\in S_{\text{train}}} (\mathbf{X}_{ji} - v_j^T a_i)^2$$

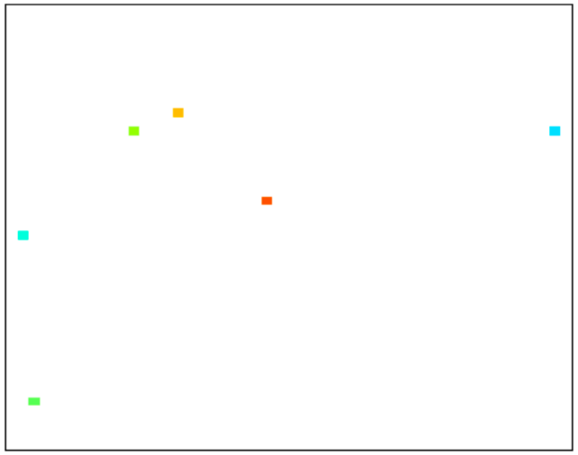
- this can be solved, for example via gradient descent or alternating minimization
- this can be quite accurate, with small number of samples

Example: 2000×2000 rank-8 random matrix

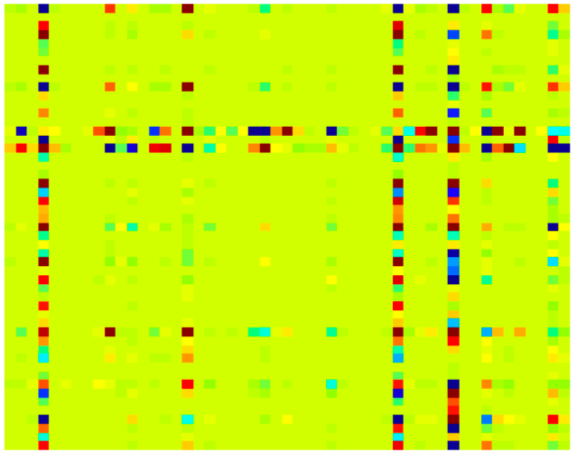
low-rank matrix $\mathbf{X}$



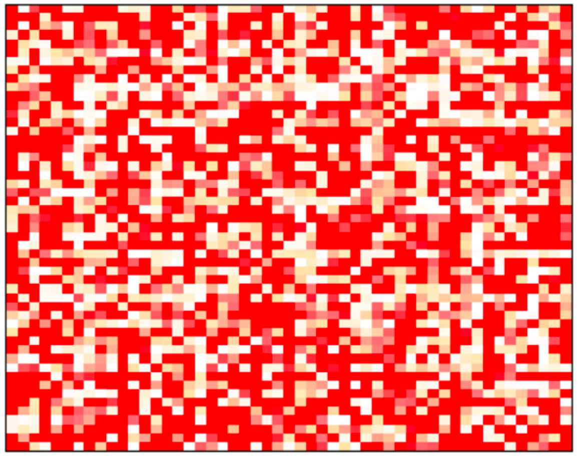
sampled matrix



Gradient descent output $\mathbf{UA}$



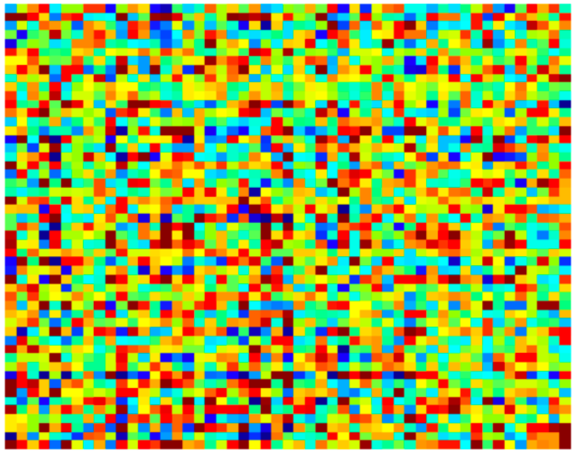
squared error $(\mathbf{X}_{ji} - (\mathbf{UA})_{ji})^2$



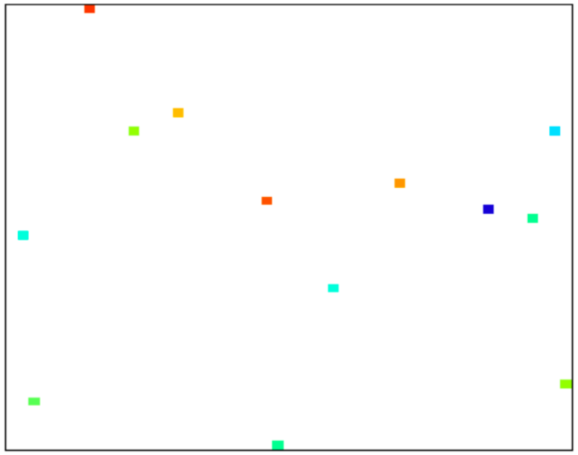
0.25% sampled

Example: 2000×2000 rank-8 random matrix

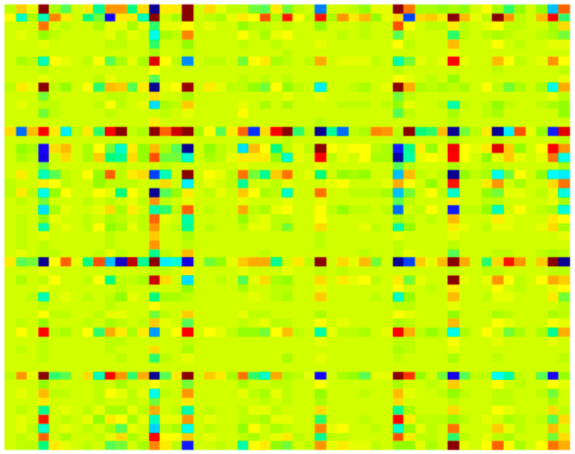
low-rank matrix $\mathbf{X}$



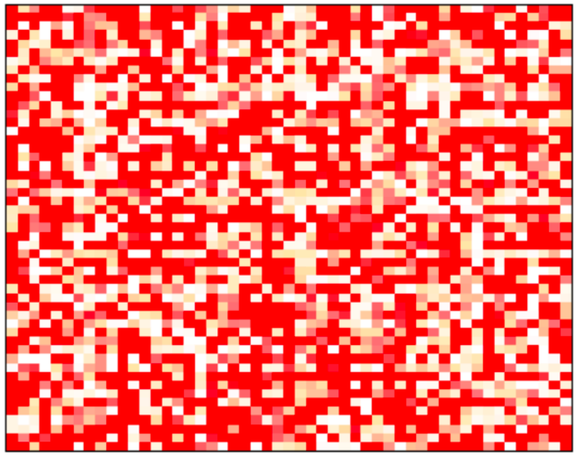
sampled matrix



Gradient descent output $\mathbf{UA}$



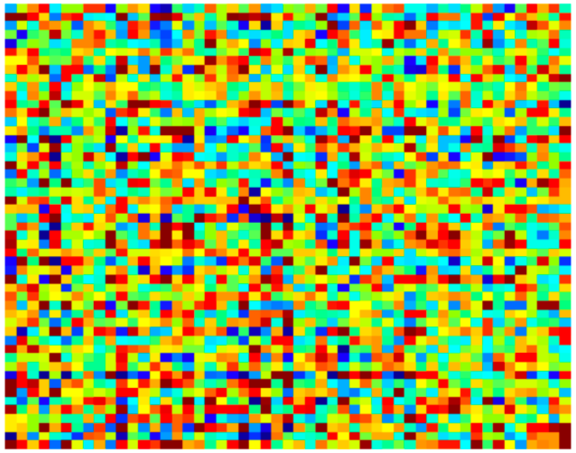
squared error $(\mathbf{X}_{ji} - (\mathbf{UA})_{ji})^2$



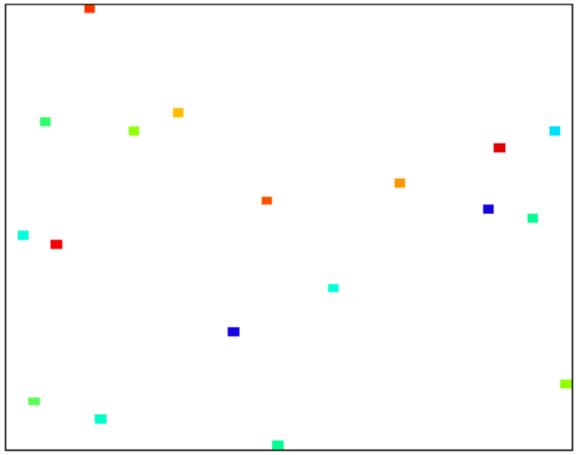
0.50% sampled

Example: 2000×2000 rank-8 random matrix

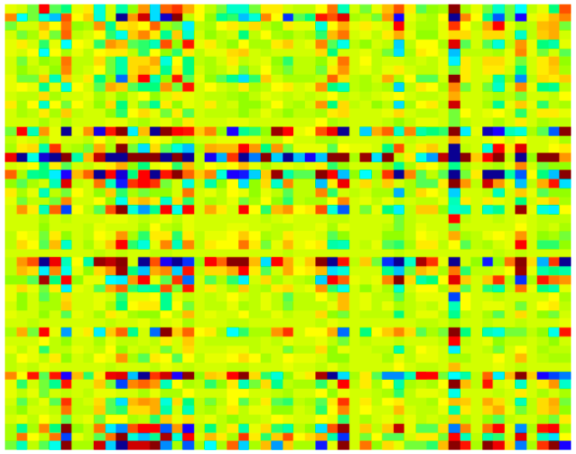
low-rank matrix $\mathbf{X}$



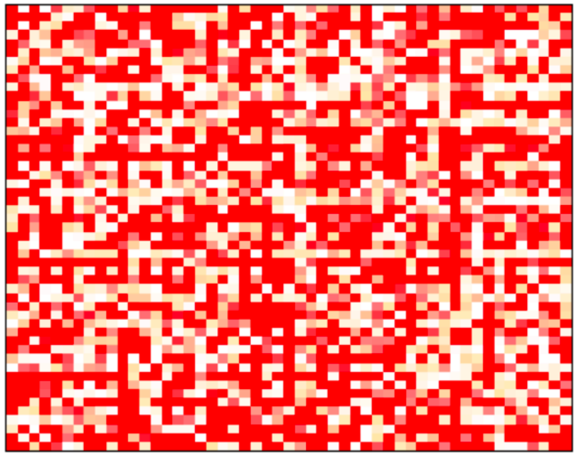
sampled matrix



Gradient descent output $\mathbf{UA}$



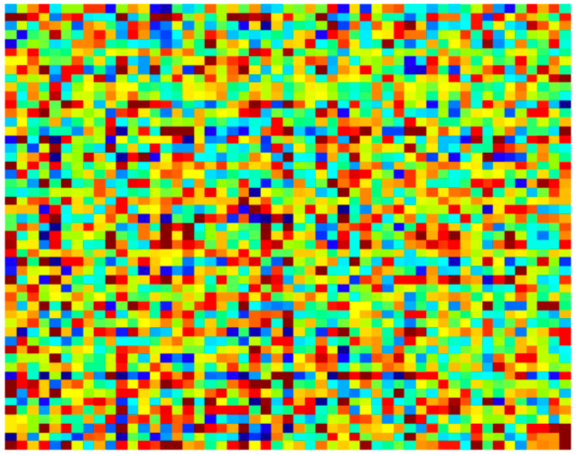
squared error $(\mathbf{X}_{ji} - (\mathbf{UA})_{ji})^2$



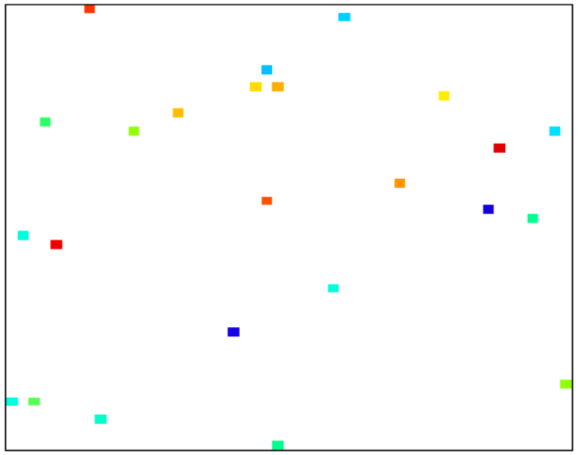
0.75% sampled

Example: 2000×2000 rank-8 random matrix

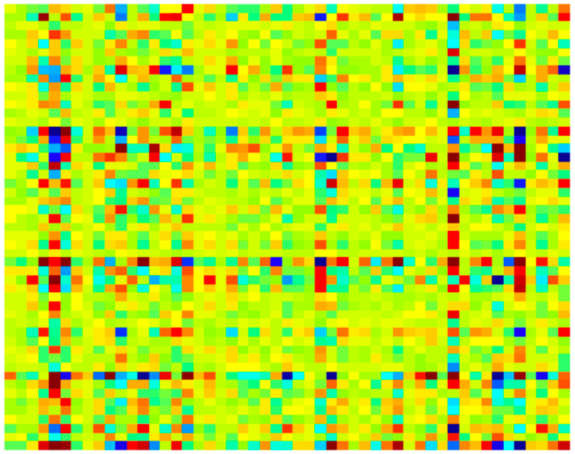
low-rank matrix $\mathbf{X}$



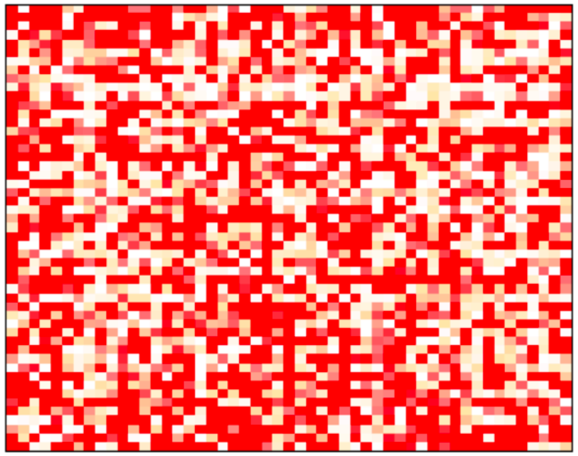
sampled matrix



Gradient descent output $\mathbf{UA}$



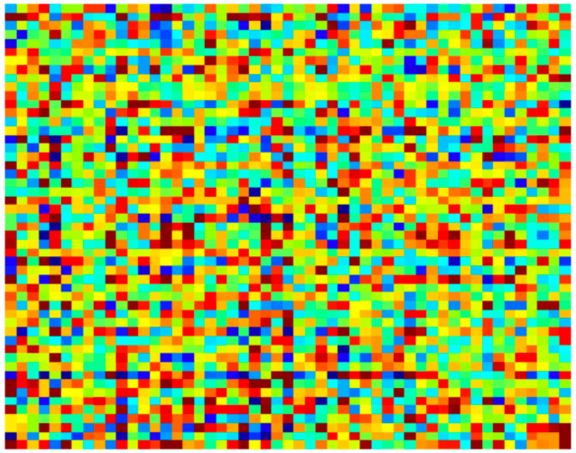
squared error $(\mathbf{X}_{ji} - (\mathbf{UA})_{ji})^2$



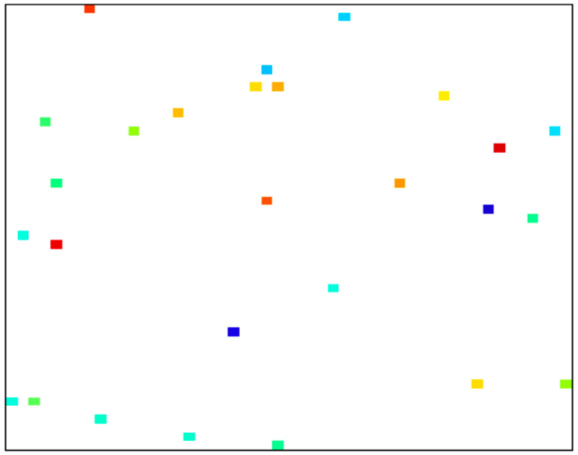
1.00% sampled

Example: 2000×2000 rank-8 random matrix

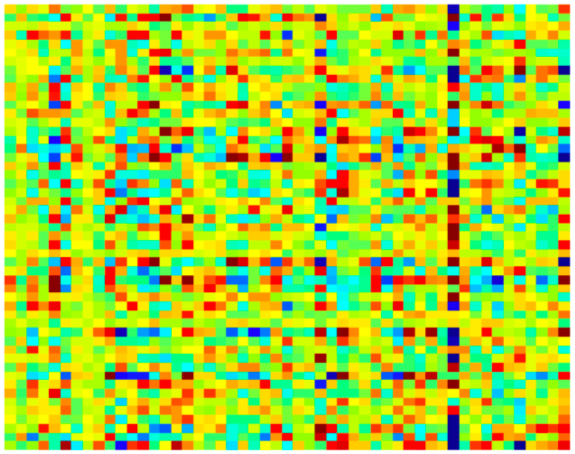
low-rank matrix $\mathbf{X}$



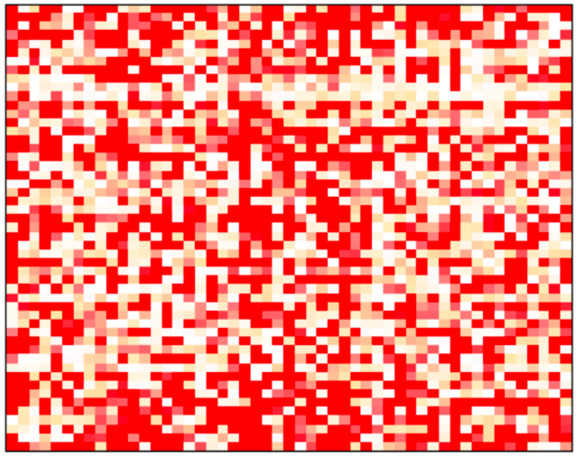
sampled matrix



Gradient descent output $\mathbf{UA}$



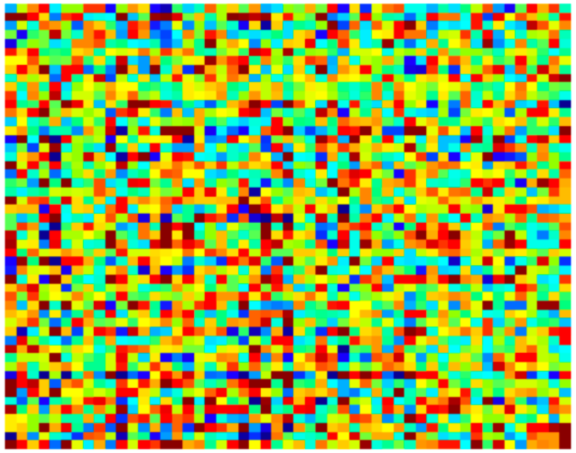
squared error $(\mathbf{X}_{ji} - (\mathbf{UA})_{ji})^2$



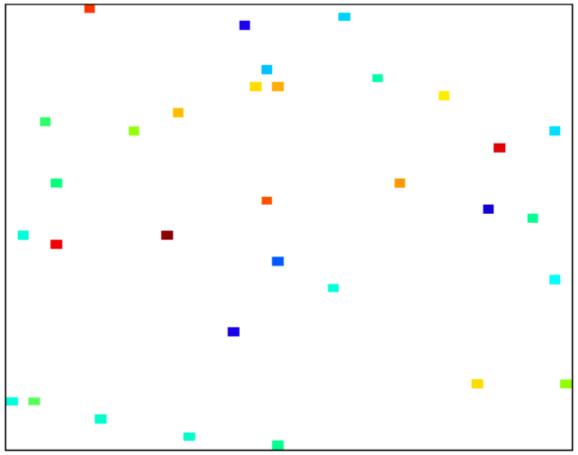
1.25% sampled

Example: 2000×2000 rank-8 random matrix

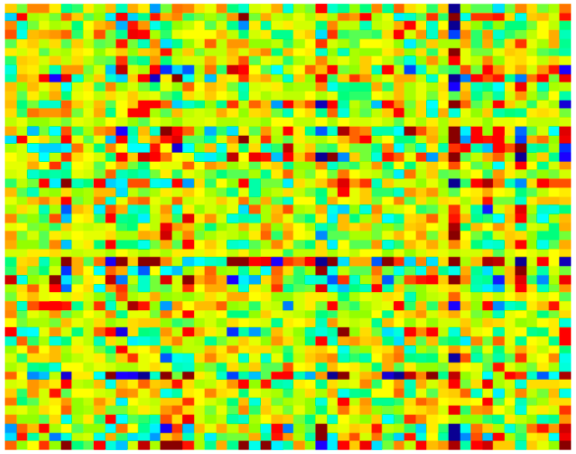
low-rank matrix $\mathbf{X}$



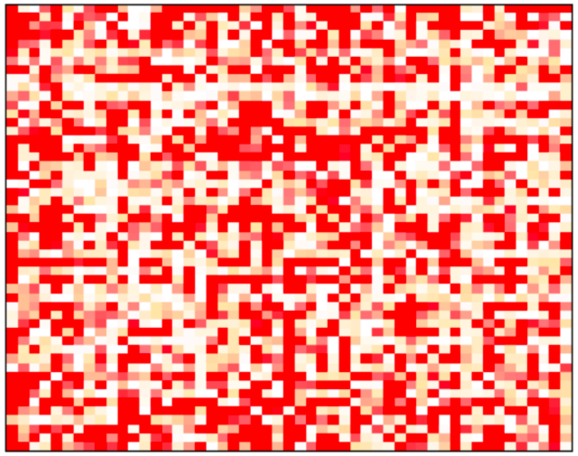
sampled matrix



Gradient descent output $\mathbf{UA}$



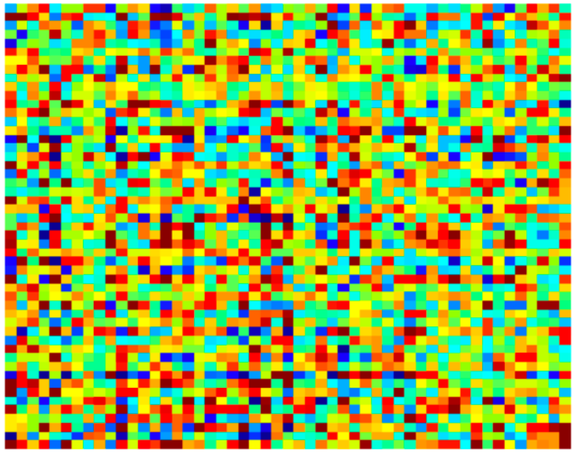
squared error $(\mathbf{X}_{ji} - (\mathbf{UA})_{ji})^2$



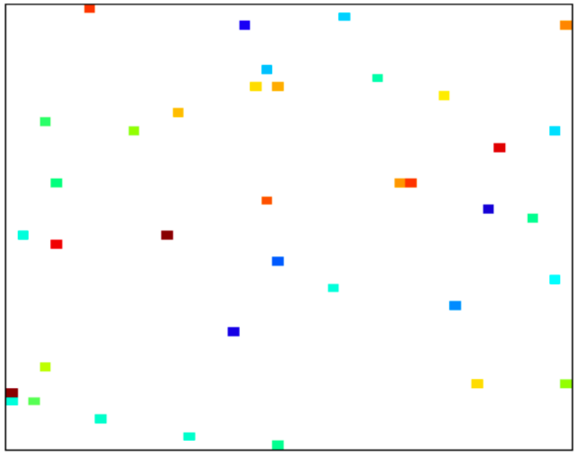
1.50% sampled

Example: 2000×2000 rank-8 random matrix

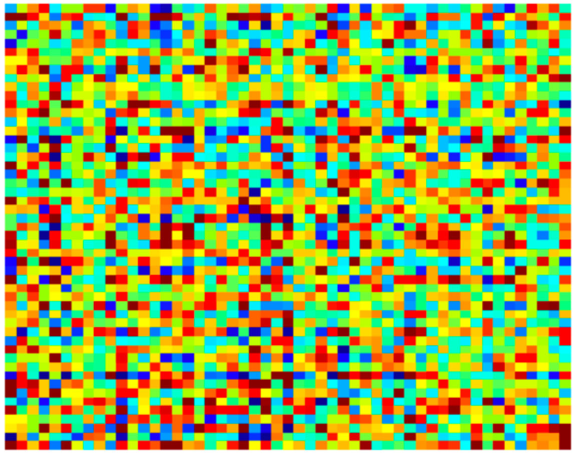
low-rank matrix $\mathbf{X}$



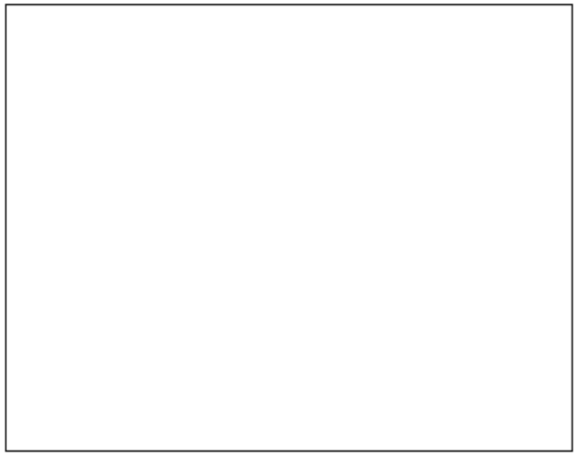
sampled matrix



Gradient descent output $\mathbf{UA}$



squared error $(\mathbf{X}_{ji} - (\mathbf{UA})_{ji})^2$



1.75% sampled

Questions?