Introduction to Data Science
Lecture 9
Unsupervised Learning

EMSE 6992 Fall 2018

Benjamin Harvey

Outline

- Unsupervised Learning
  - K-Means clustering
  - DBSCAN
  - Matrix Factorization
- Performance

Machine Learning

- **Supervised:** We are given input/output samples (X, y) which we relate with a function y = f(X). We would like to "learn" f, and evaluate it on new data. Types:
  - **Classification:** y is discrete (class labels).
  - **Regression:** y is continuous, e.g. linear regression.

- **Unsupervised:** Given only samples X of the data, we compute a function f such that y = f(X) is "simpler".
  - **Clustering:** y is discrete
  - Y is continuous: **Matrix factorization, Kalman filtering, unsupervised neural networks**.

Machine Learning

- **Unsupervised:**
  - Cluster some hand-written digit data into 10 classes.
  - What are the top 20 topics in Twitter right now?
  - Find and cluster distinct accents of people at Berkeley.

Techniques

- **Supervised Learning:**
  - kNN (k Nearest Neighbors)
  - Naïve Bayes
  - Linear + Logistic Regression
  - Support Vector Machines
  - Random Forests
  - Neural Networks
- **Unsupervised Learning:**
  - **Clustering**
  - **Matrix Factorization**
  - HMMs (Hidden Markov Models)
  - Neural Networks

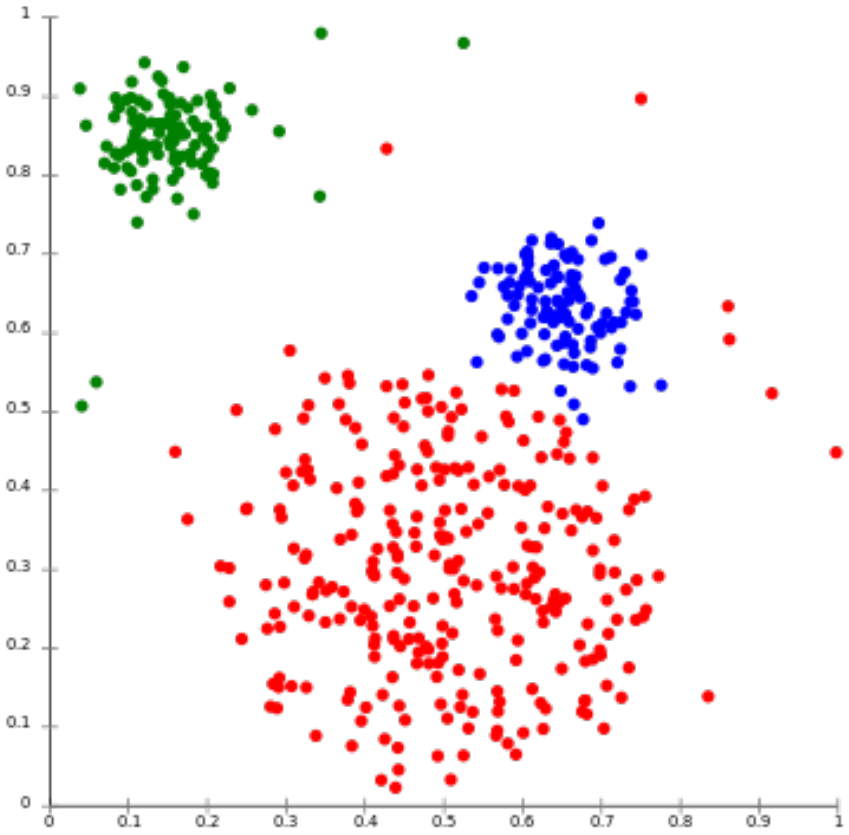
Clustering – Why?

Examples:

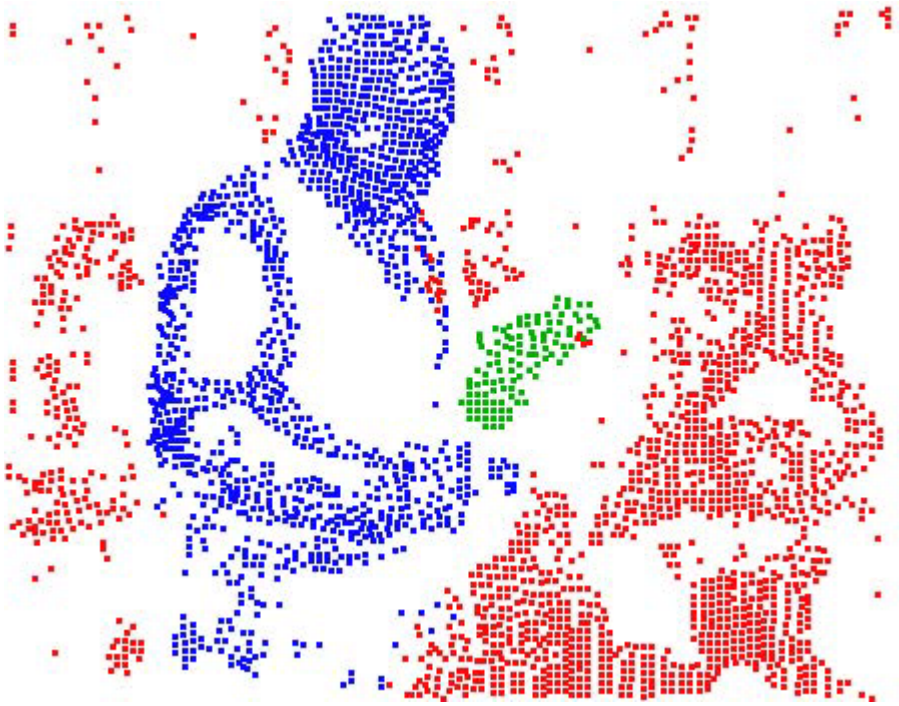
- Segment: image segmentation
- Compression: Cluster-based kNN, e.g. handwritten digit recognition.
- Underlying process: Accents of people at Berkeley (??) – because place of origin strongly influences the accent you have.

Stereotypical Clustering

Note: Points are samples plotted in feature space, e.g. 10,000-dimensional space for 100x100 images.

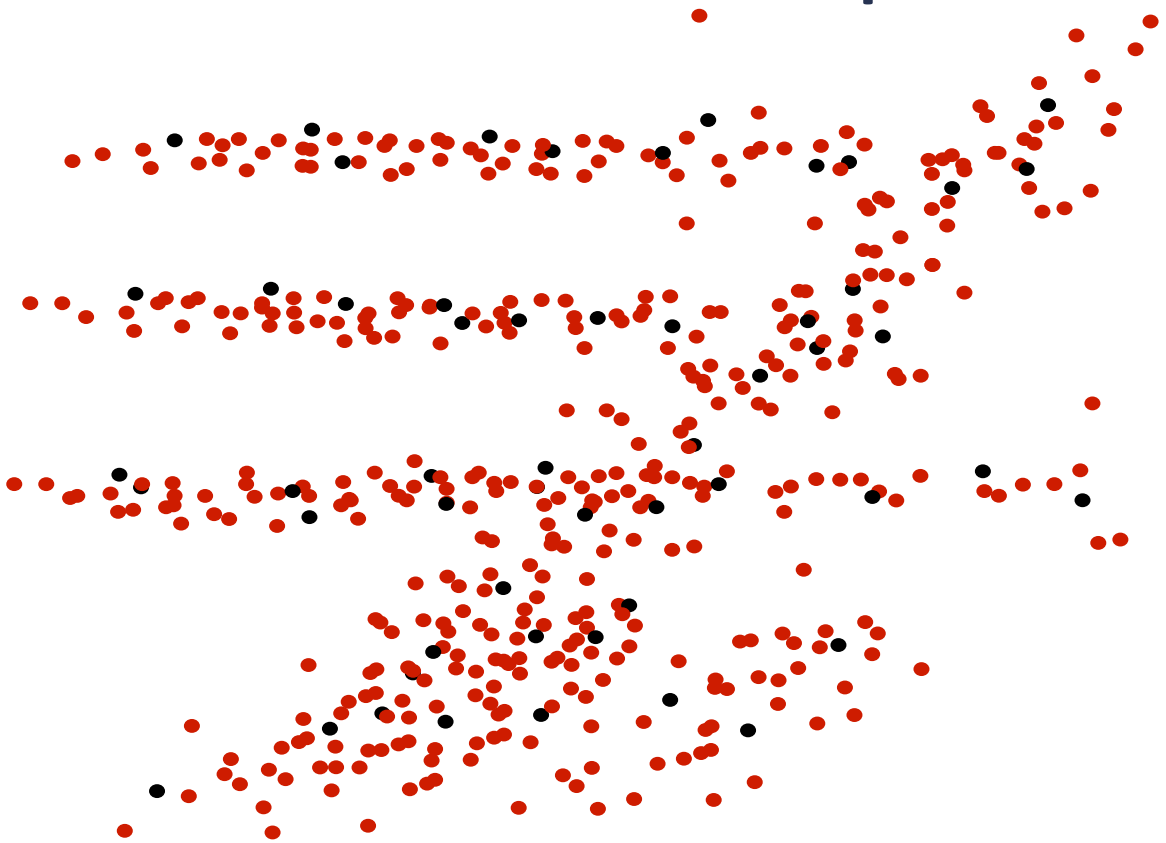


Clustering for Segmentation



Note: We break an image into regions of points with similar features (Brox and Malik, ECCV 2010).

Condensation/Compression



We don’t claim that clusters model underlying structure, but that they give a reduced-variance “sample” of the data.

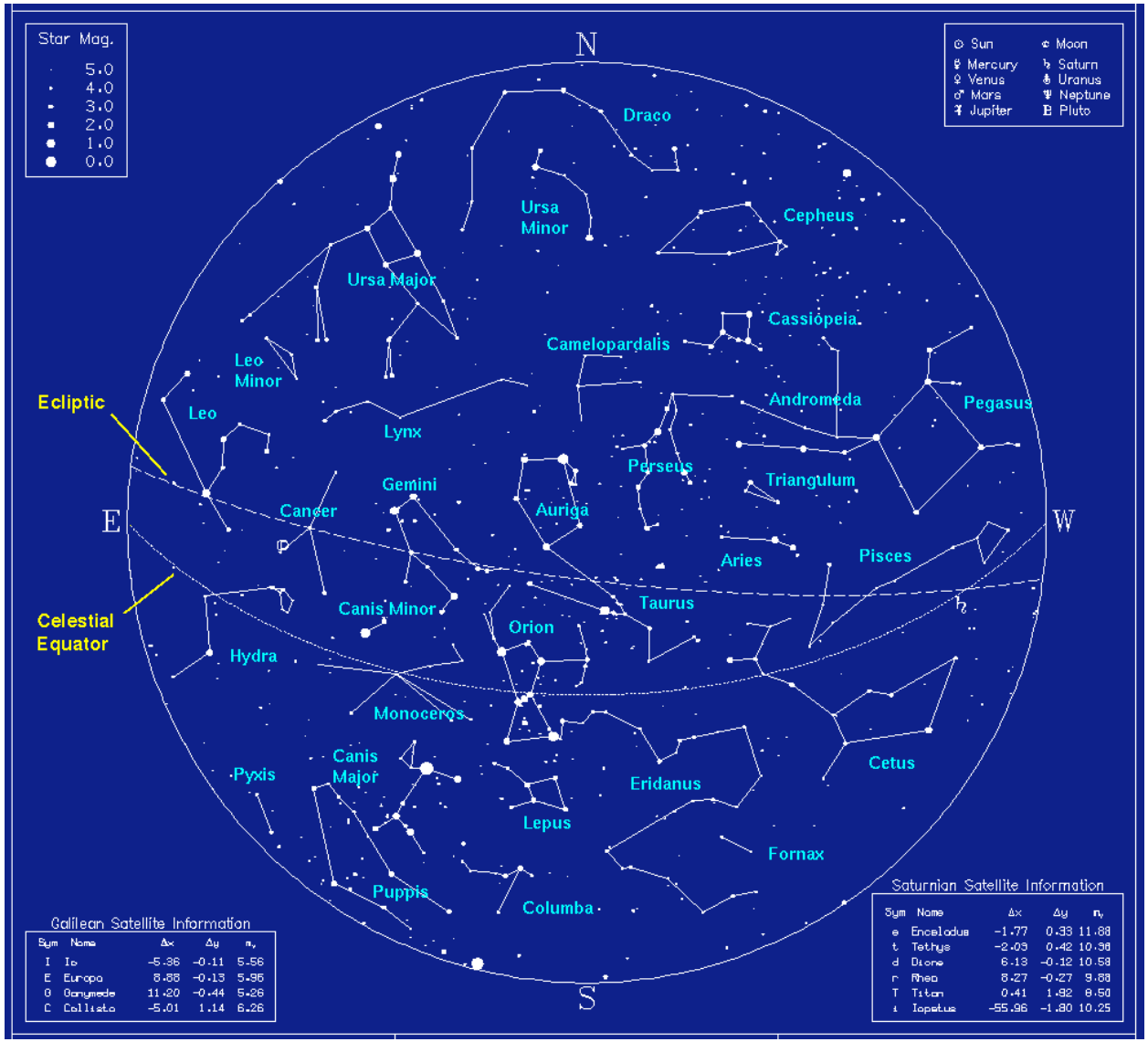
“Cluster Bias”

- Human beings conceptualize the world through categories represented as *examplars* (Rosch 73, Estes 94).



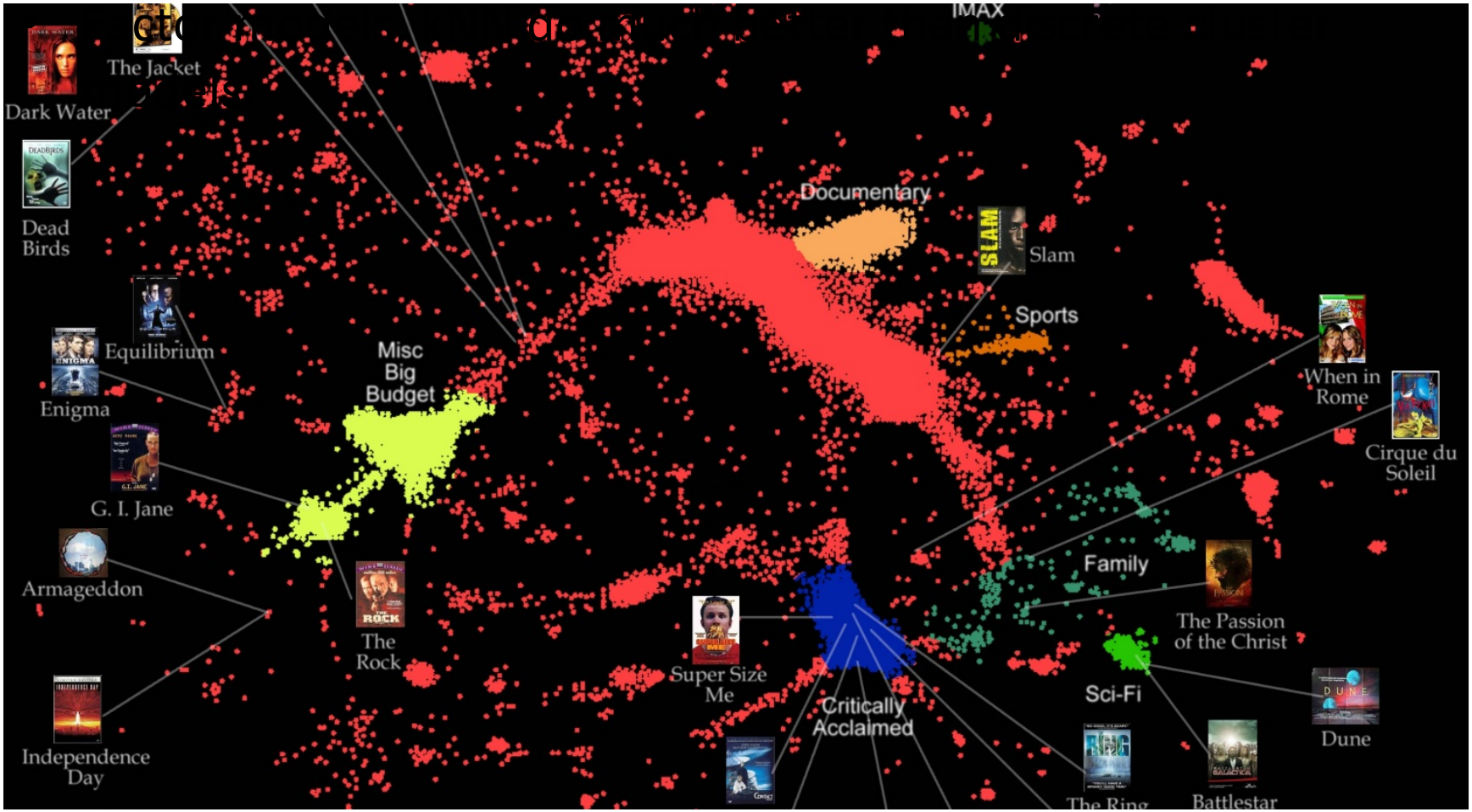
- We tend to see cluster structure whether it is there or not.
- Works well for dogs, but…

Cluster Bias



Netflix

- More of a continuum than discrete clusters



“Cluster Bias”

Upshot:

- **Clustering is used more than it should be**, because people assume an underlying domain has discrete classes in it.
- This is especially true for characteristics of people, e.g. Myers-Briggs personality types like “ENTP”.
- In reality the underlying data is usually **continuous**.
- Just as with Netflix, continuous models (matrix factorization, “soft” clustering, kNN) tend to do better.

Terminology

- **Hierarchical clustering:** clusters form a hierarchy. Can be computed bottom-up or top-down.
- **Flat clustering:** no inter-cluster structure.

- **Hard clustering:** items assigned to a unique cluster.
- **Soft clustering:** cluster membership is a real-valued function, distributed across several clusters.

K-means clustering

The standard k-means algorithm is based on **Euclidean distance**.

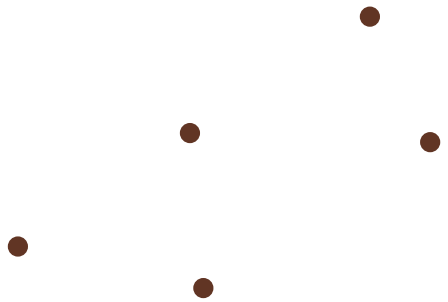
The cluster quality measure is an **intra-cluster measure only**, equivalent to the sum of item-to-centroid kernels.

A simple greedy algorithm locally optimizes this measure (usually called Lloyd's algorithm):

- **Find the closest cluster center** for each item, and assign it to that cluster.
- **Recompute the cluster centroid** as the mean of items, for the newly-assigned items in the cluster.

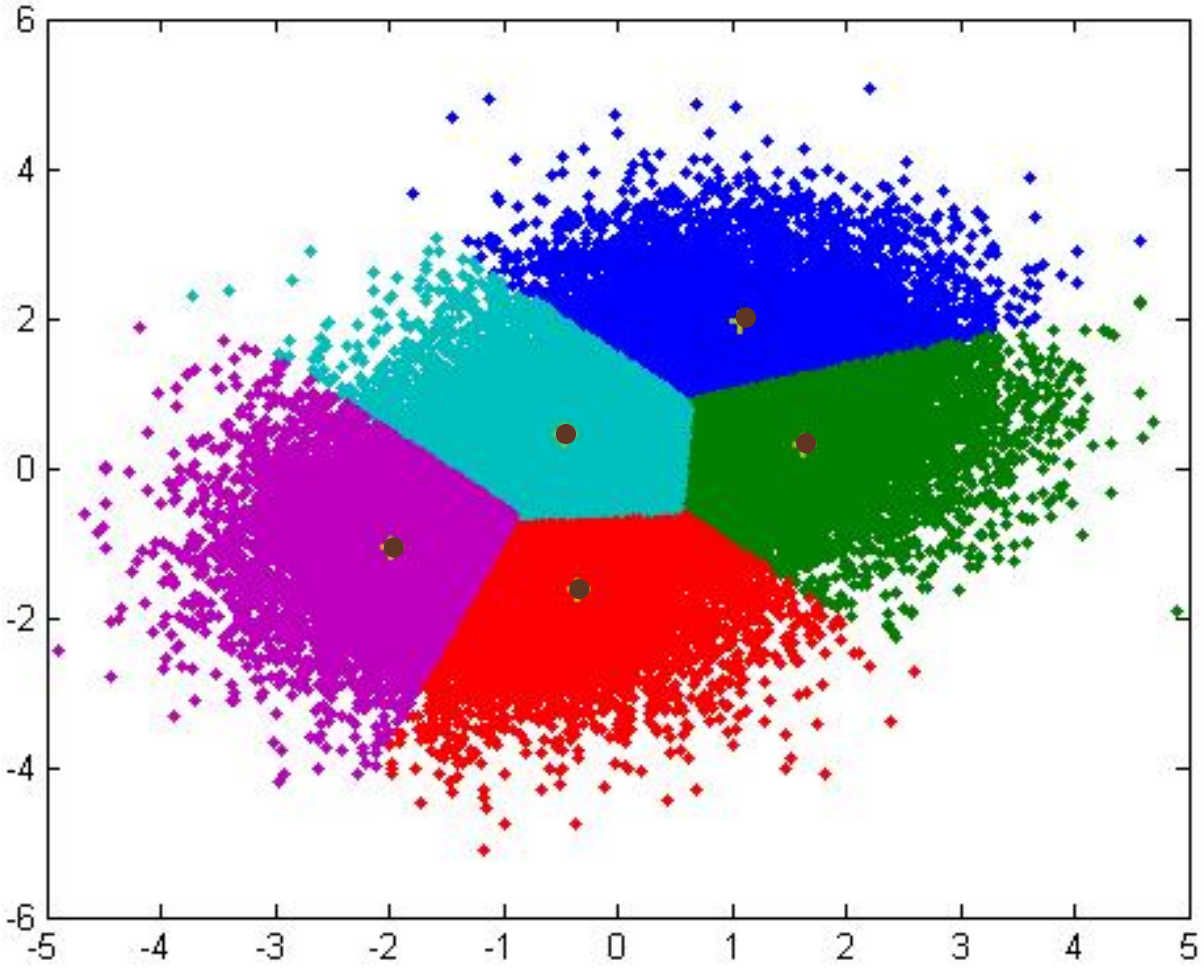
K-means clustering

Cluster centers – can pick by sampling the input data.



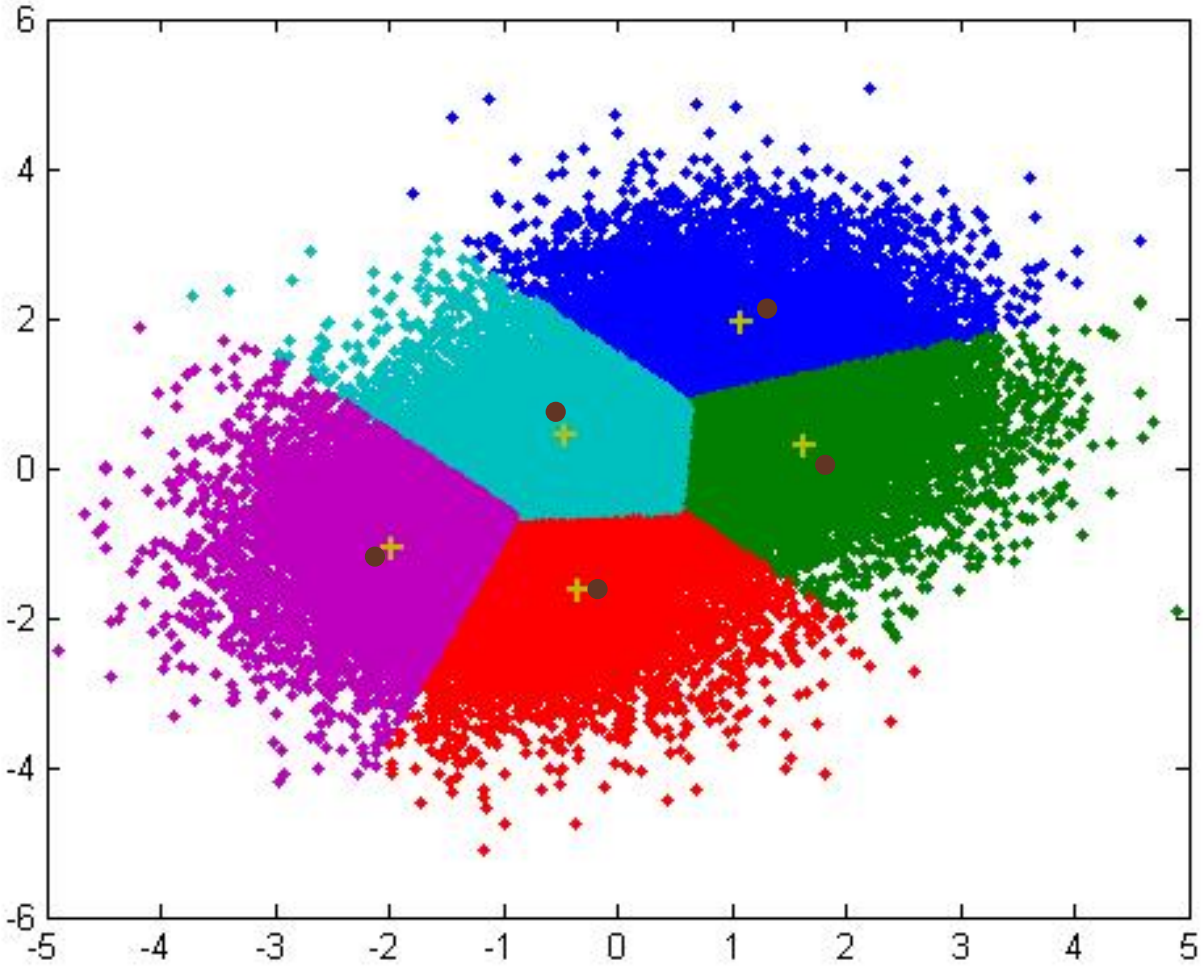
K-means clustering

Assign points to closest center



K-means clustering

Recompute centers (old = cross, new = dot)



K-means clustering

Iterate:

- For fixed number of iterations
- Until no change in assignments
- Until small change in quality

K-means Initialization

We need to pick some points for the first round of the algorithm:

- **Random sample:** Pick a random subset of k points from the dataset.
- **K-Means++:** Iteratively construct a random sample with good spacing across the dataset.

Note: Finding an exactly-optimal k-Means clustering is NP-hard. Randomization helps avoid bad configurations.

K-means++

Start:

- Choose first cluster center at random from the data points

Iterate:

- For every remaining data point x, compute D(x) the distance from x to the closest cluster center.
- Choose a remaining point x randomly with probability proportional to $D(x)^2$, and make it a new cluster center.

Intuitively, this finds a sample of widely-spaced points avoiding "collapsing" of the clustering into a few internal centers.

K-means properties

- It's a greedy algorithm with random setup – **solution isn't optimal** and varies significantly with different initial points.
- Very simple convergence proofs.
- **Performance is O(nk) per iteration**, not bad and can be heuristically improved.
  n = total features in the dataset, k = number clusters
- Many generalizations, e.g.
  - Fixed-size clusters
  - Simple generalization to m-best soft clustering
- As a "local" clustering method, it works well for data condensation/compression.

Choosing clustering dimension

- AIC or Akaike Information Criterion:

$$\textbf{AIC:} \qquad K = \arg\min_{K}[-2L(K) + 2q(K)]$$

- K=dimension, L(K) is the likelihood (could be RSS) and q(K) is a measure of model complexity (cluster description complexity).

- AIC favors more compact (fewer clusters) clusterings.

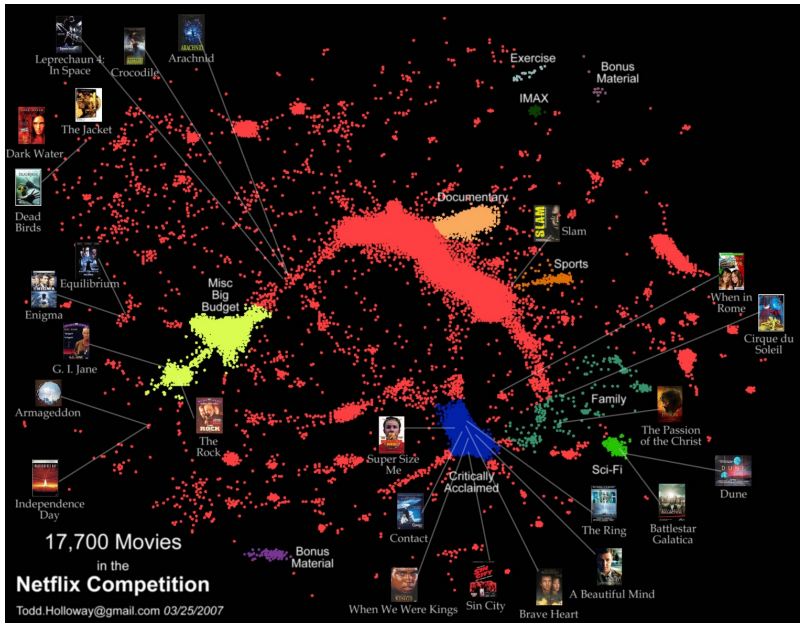
- For sparse data, AIC will incorporate the number of non-zeros in the cluster spec. Lower is better.

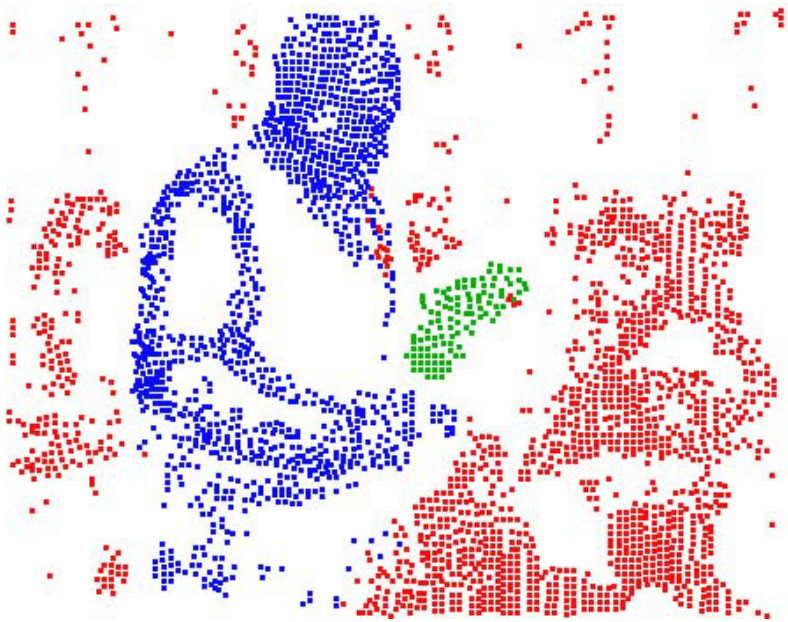
Outline

- Unsupervised Learning
  - K-Means clustering
  - DBSCAN
  - Matrix Factorization
- Performance

DBSCAN

- Motivation: Centroid-based clustering methods like k-Means favor clusters that are spherical, and have great difficulty with anything else.
- But with real data we have:





DBSCAN

- DBSCAN performs density-based clustering, and follows the shape of dense neighborhoods of points.

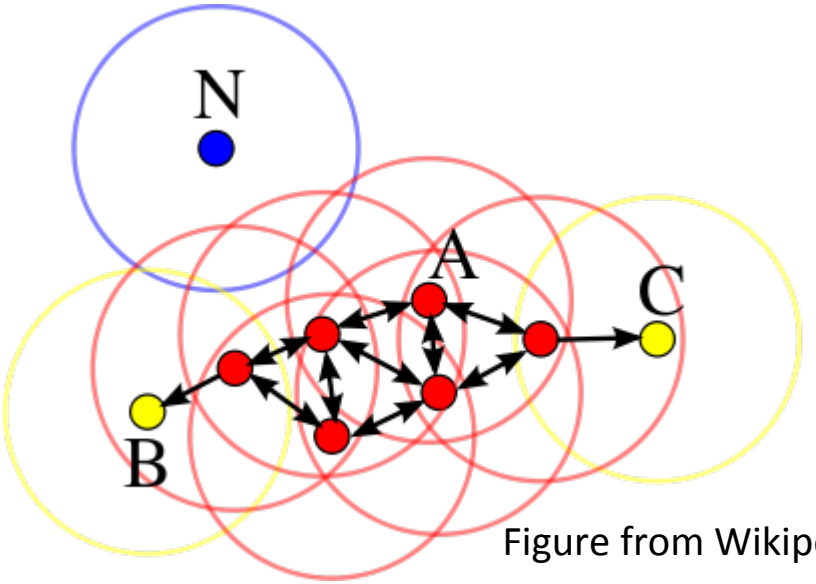
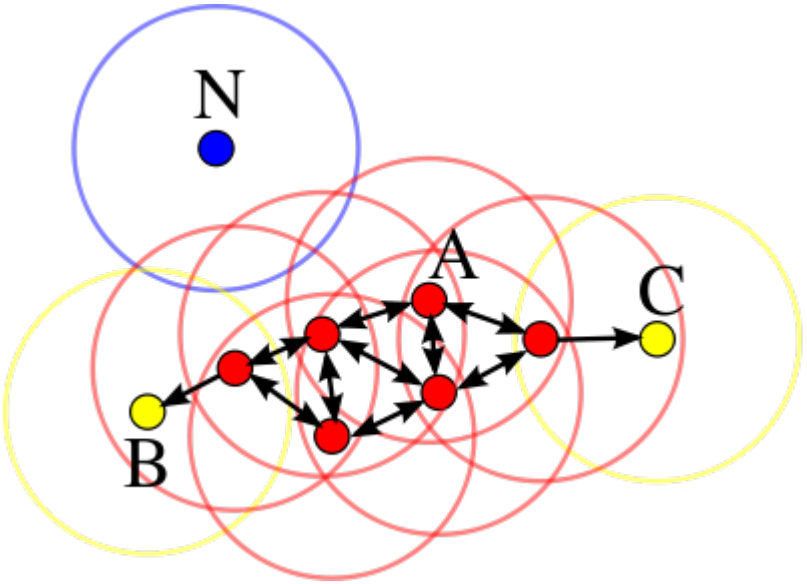


Figure from Wikipedia Article on DBSCAN

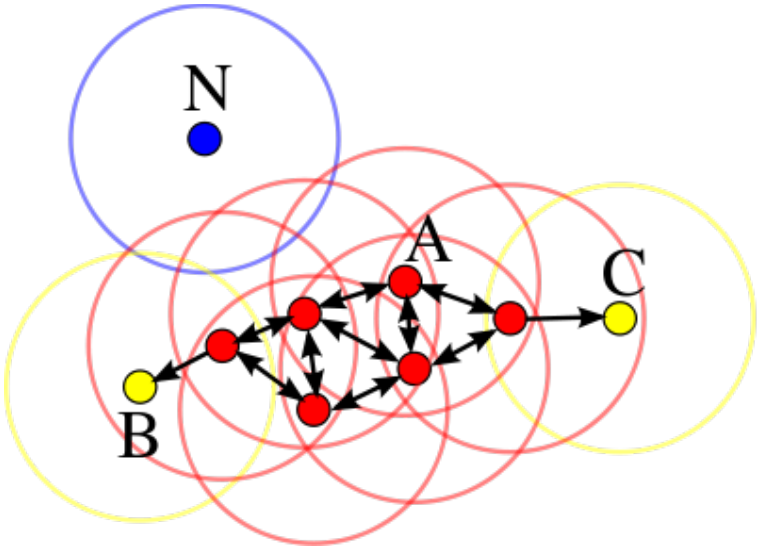
- **Core points** have at least minPts neighbors in a sphere of diameter ε around them.
- The red points here are core points with at least minPts = 3 neighbors in an ε-sphere around them.

DBSCAN



- **Core points** can **directly reach** neighbors in their ε-sphere.
- Non-core points cannot directly reach another point.
- Point q is **density-reachable** from p if there is a series of points $p=p_1,p_2,\ldots p_n=q$ s.t. p_{i+1} is directly reachable from p_i
- All points not density-reachable from any other points are **outliers**.

DBSCAN Clusters



- Points, p, q are **density-connected** if there is a point o such that both p and q are density-reachable from o.
- A **cluster** is a set of points which are **mutually density-connected**.
- If a point is density-reachable from a cluster-point, it is part of the cluster as well.
- For the figure above, points A are density-connected, B, C are density reachable and N is not.

DBSCAN Algorithm

```
DBSCAN(D, eps, MinPts) {
   C = 0
   for each point P in dataset D {
      if P is visited
         continue next point
      mark P as visited
      NeighborPts = regionQuery(P, eps)
      if sizeof(NeighborPts) < MinPts
         mark P as NOISE
      else {
         C = next cluster
         expandCluster(P, NeighborPts, C, eps, MinPts)
      }
   }
}
```

Source Wikipedia article on DBSCAN

DBSCAN Performance

- DBSCAN uses all-pairs point distances, but using an efficient indexing structure (and assuming there are not too many neighbors), each neighborhood query takes O(log n) time.
- The algorithm overall can be made to run in **O(n log n)**.
- Fast neighbor search becomes progressively harder (higher constants) in higher dimensions.
- The fast implementation also assumes data can be held in memory.
- The fall-back is an $O(n^2)$ out-of-memory implementation.

5-minute break

Outline

- Unsupervised Learning
  - K-Means clustering
  - DBSCAN
  - Matrix Factorization
- Performance

Matrix Factorization - Motivation

- For problems with linear relationships between a response *Y* and feature data *X* we have the linear regression formula:

$$\hat{Y} = \hat{\beta}_0 + \sum_{j=1}^{p} X_j \hat{\beta}_j$$

- But as the dimension of *Y* grows, we quickly "run out of data" – the model has more degrees of freedom than we have data.
  - e.g. recommendations, information retrieval
  - Features and responses are the same, e.g. ratings of products, or relevance of a term to a document.
  - We have to deal with thousands or millions of responses, as well as thousands or millions of features.

- This is one flavor of the "curse of dimensionality".

Matrix Factorization - Motivation

- High-dimensional data will often have simpler structure, e.g.
  - Topics in documents
  - General product characteristics (reliability, cost etc.)
  - User preferences
  - User personality, demographics
  - Dimensions of sentiment
  - Themes in discussions
  - “Voices” or styles in documents
- We can approximate these effects using a lower-dimensional model, which we assume for now is linear.

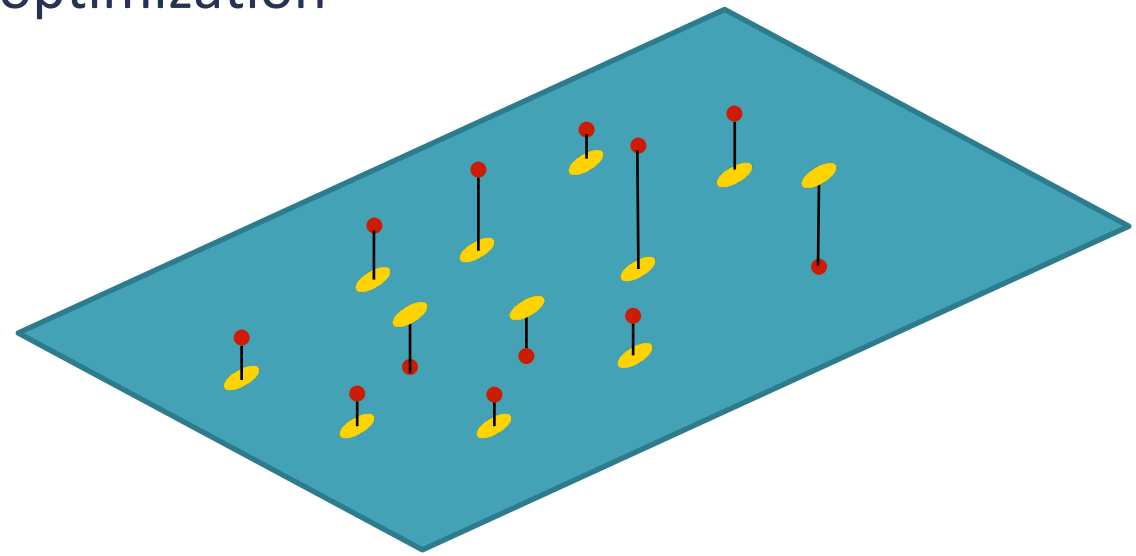
Matrix Factorization - Motivation

The subspace allows us to predict the values of other features from known features:

d coordinates define a unique point on a d-dimensional plane.

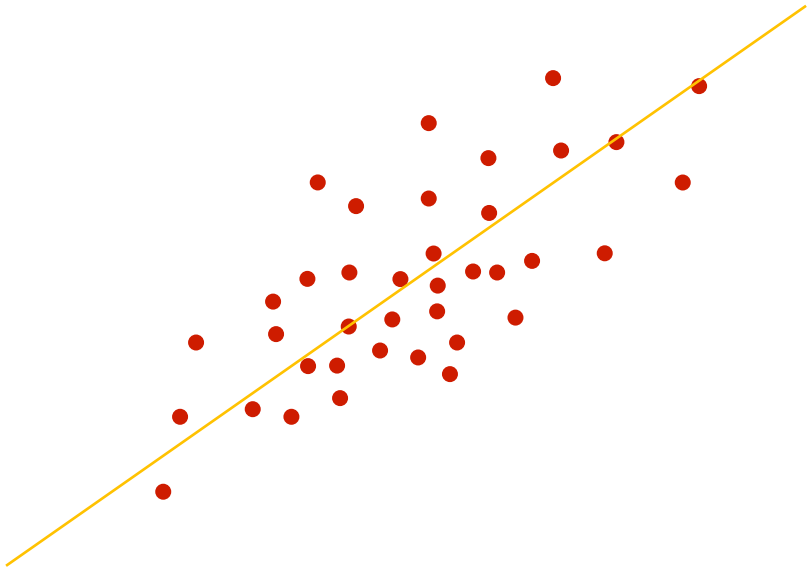
From these measurements, we can predict other coordinates.

Applications: User product preferences, CTR prediction, search engine optimization



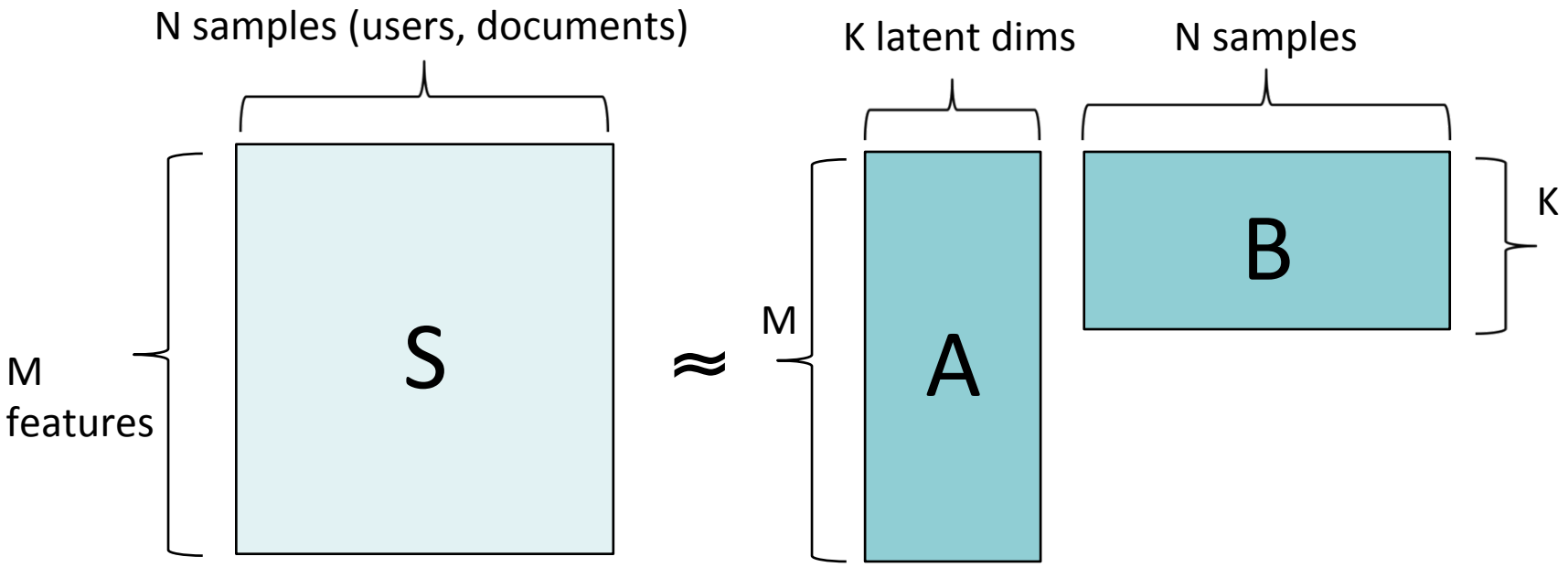
Matrix Factorization - Motivation

Usually, data vary more strongly in some dimensions than others. Fitting a linear model to the (k) strongest directions gives the best approximation to the data in a least-squares sense.



Matrix Factorization

Algebraically, we have a high-dimensional data matrix (typically sparse) S, which we approximate as a product of two dense matrices A and B with low "inner" dimension:



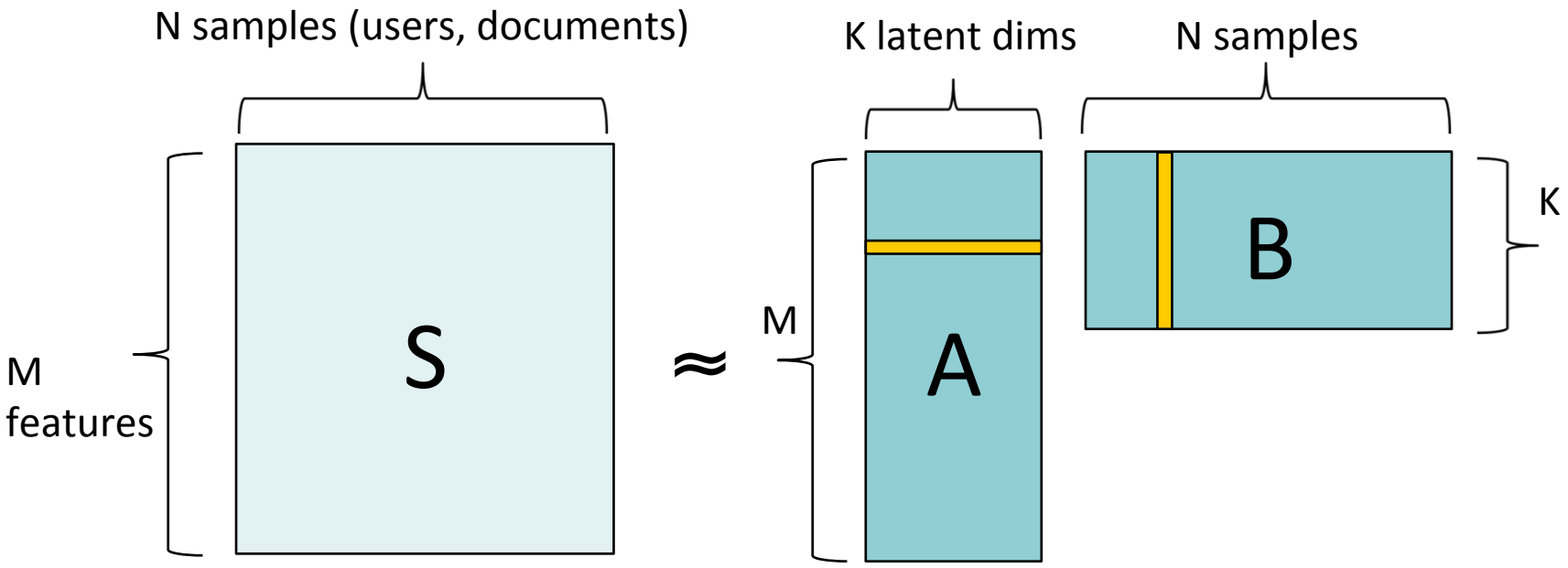
From the factorization, we can fill in missing values of S. This problem is often called "Matrix Completion".

Matrix Factorization

Columns of B represent the distribution of topics the n^{th} sample.

Rows of A represent the distribution of topics in the m^{th} feature.

These weights allow us to interpret the latent dimensions.



Matrix Factorization

is a popular method for sparse, linear data (ratings, kw/URL combos, CTR on ads).

- Input matrix S is sparse, and we approximate it as a low-dimensional product: $S \approx A * B$. Zeros of S encode **don't know** state.

- Typically minimize quadratic loss on **non-zeros (actual observations)** of S

$$L_2(S - A *_S B)$$

and with L_2 regularizers on A and B. Here $*_S$ denotes the product evaluated only at non-zeros of S.

Matrix Factorization with SGD

We can compute gradients with respect to A and B which are:

$$\frac{dl}{dA} = -2(S - A *_S B)B^T + 2w_A A$$

where w_A is the weighting of the regularizer on A, and

$$\frac{dl}{dB} = -2A^T(S - A *_S B) + 2w_B B$$

where w_B is the weight of the regularizer on B.

Then the loss can be minimized with SGD. This method is quite fast, but suffers from weak local optima.

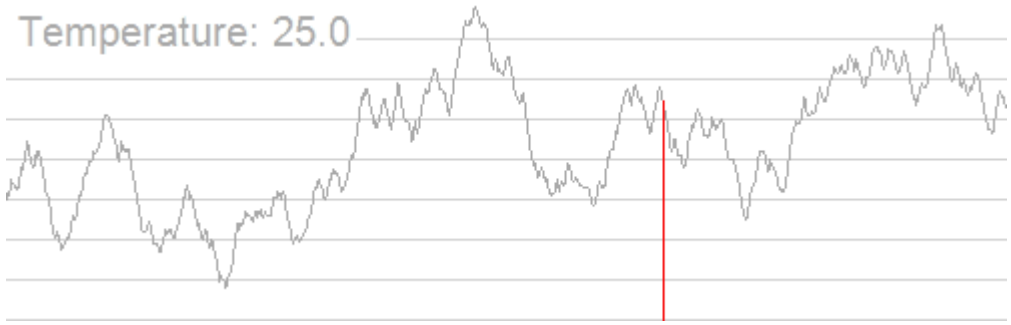
Matrix Factorization with MCMC

The local optima problem can be minimized by using MCMC methods (Markov-Chain Monte-Carlo).

Instead of moving only in the direction of the gradient, these methods sometimes move “down” which allows them to jump into other optima.

Changing the “energy” of the search allows “annealing” which generally finds good local optima.

This is probably the most accurate method for matrix factorization right now.



Alternating Least Squares (ALS)

Given A, solving for a column of B is a standard linear regression task, and has a closed form:

$$B_i = (w_B I - A^T D_i A)^{-1} A^T S_i$$

where S_i is the i^{th} column of S, D_i is a diagonal matrix with ones at the non-zeros rows of S_i.

A similar formula exists for rows of A.

Because of the matrix inversions, this method is quite expensive requiring $O(K^3(N+M))$ time. This is however a popular implementation in cloud systems (e.g. Spark, Powergraph, Mahout).

Netflix

- From the first year onwards all competitive entries used a dimension reduction model.
- The dimension reduction accounts for about half the algorithms' gain over the baseline.
- The winning team's model is roughly:

$$\hat{r}_{ui} = b_{ui} + q_i^T \left(|\mathrm{R}(u)|^{-\frac{1}{2}} \sum_{j \in \mathrm{R}(u)} (r_{uj} - b_{uj}) x_j + |\mathrm{N}(u)|^{-\frac{1}{2}} \sum_{j \in \mathrm{N}(u)} y_j \right)$$

- r is the rating, b is a baseline, q is an item factor, the remainder is the "user factor".
- It incorporates new factor vectors x: for explicit ratings and y: for implicit ratings.

Netflix

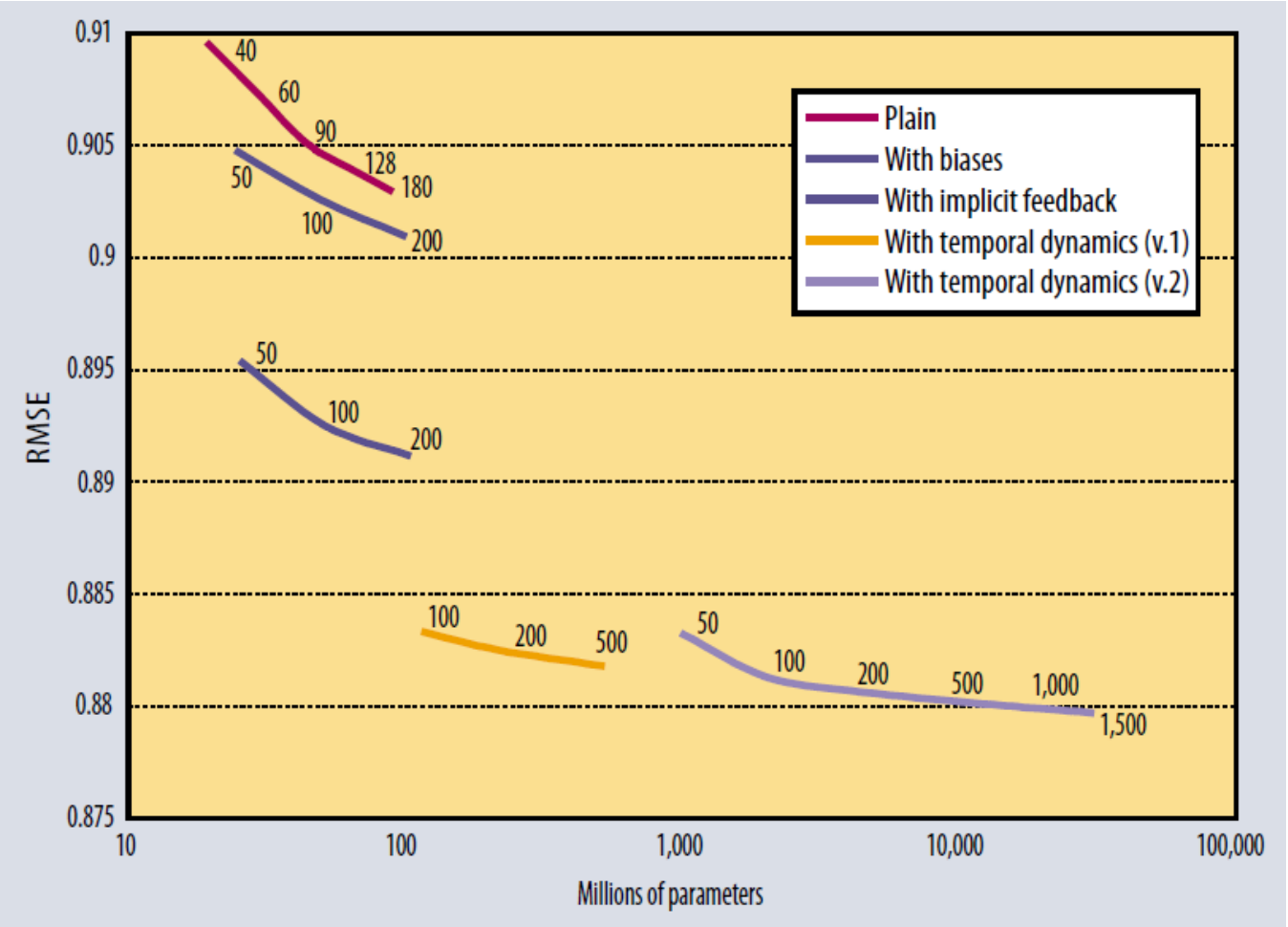
- Summary of dimension reduction models:

Model	50 factors	100 factors	200 factors
SVD	0.9046	0.9025	0.9009
Asymmetric-SVD	0.9037	0.9013	0.9000
SVD++	0.8952	0.8924	0.8911

- Dimension reduction augmented by neighborhoods:

	50 factors	100 factors	200 factors
RMSE	0.8877	0.8870	0.8868
time/iteration	17min	20min	25min

Netflix Performance



From Koren, Bell, Volinksy, "Matrix Factorization Techniques for Recommender Systems" IEEE Computer 2009.

Outline

- Unsupervised Learning
  - K-Means clustering
  - DBSCAN
  - Matrix Factorization
- Performance

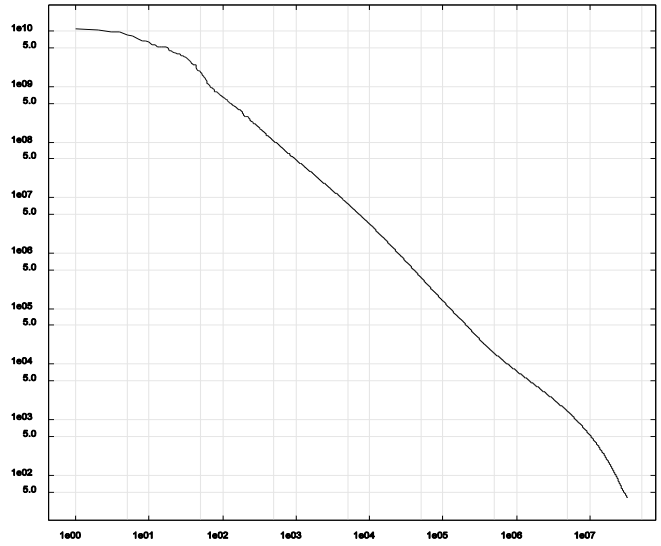
Performance

There are several reasons to design for performance:

- **Dataset size** (assuming model quality improves with size)
- **Model size** (bigger models generally perform better)

Both the above are true (but not obvious) for power-law datasets. Because the data "tail" is long, models also need to be long-tailed: it allows you to model less-frequent users and features.

Training long-tailed models requires more data.



Offline/Online Performance

There are two different dimensions of performance:

1. **Offline: Model training**. Plenty of resources. Typical goal is 1 to few days training time. Best possible model accuracy.

2. **Online: Model prediction**. Limited resources (memory and machines). Goal is usually to minimize latency, and perhaps online model size.

2. Influences 1. Models trained offline have to fit and be fast in the deployment environment.

Three Approaches to Performance

There are at least three approaches to improving performance:

- **Algorithmic improvements**. e.g. adaptive SGD methods.

- **Computational improvements**: Fully optimize code, leverage hardware (e.g. Graphics Processing Units).

- **Cluster scale-up**: Distribute model or data across computers on a network.

Three Approaches to Performance

There are at least three approaches to improving performance:

- **Algorithmic improvements**. e.g. adaptive SGD methods.
- **Free!** Orders of magnitude improvements possible
- **Computational improvements**: Fully optimize code, leverage hardware (e.g. Graphics Processing Units).

- **Cluster scale-up**: Distribute model or data across computers on a network.

Three Approaches to Performance

There are at least three approaches to improving performance:

- **Algorithmic improvements**. e.g. adaptive SGD methods.
- **Free!** Orders of magnitude improvements possible
- **Computational improvements**: Fully optimize code, leverage hardware (e.g. Graphics Processing Units).
- **Free!** Orders of magnitude improvements possible. Compounds algorithmic improvements.
- **Cluster scale-up**: Distribute model or data across computers on a network.

Three Approaches to Performance

There are at least three approaches to improving performance:

- **Algorithmic improvements**. e.g. adaptive SGD methods.
- **Free!** Orders of magnitude improvements possible.
- **Computational improvements**: Fully optimize code, leverage hardware (e.g. Graphics Processing Units).
- **Free!** Orders of magnitude improvements possible. Compounds algorithmic improvements.
- **Cluster scale-up**: Distribute model or data across computers on a network.
- **Superlinear cost:** a k-fold speedup requires, fn(k) > k additional cost. Usually does not compound algorithmic and computational improvements (network bottleneck).

Three Approaches to Performance

There are at least three approaches to improving performance:

- **Algorithmic improvements**. e.g. adaptive SGD methods.
- **Free!** Orders of magnitude improvements possible.
- **Computational improvements**: Fully optimize code, leverage hardware (e.g. Graphics Processing Units).
- **Free!** Orders of magnitude improvements possible. Compounds algorithmic improvements.
- **Cluster scale-up**: Distribute model or data across computers on a network.
- **Codesign:** using knowledge of the algorithms and data (power law), we can reduce network load several orders of magnitude. Network topology awareness can improve things dramatically. Linear speedup is possible this way.