SAPIENZA
UNIVERSITÀ DI ROMA

FACULTY OF INFORMATION ENGINEERING,
INFORMATICS, AND STATISTICS

Big Data Computing

DEPARTMENT OF COMPUTER SCIENCE

Professors:
Gabriele Tolomei

Student:
Davide Pietragalla

Academic Year 2024/2025

Contents

1 Introduction

Suppose you had to determine whether a given element x exists in a collection S of N items. This is a fundamental and widely encountered problem in computer science.

Let's also suppose S is a list of N integers, and we want to check if a specific number x is in this list.

- If the list is **unsorted**, a linear scan is required, which takes $\mathcal{O}(N)$ time.
- If the list is **sorted**, a binary search can be performed, which requires only $\mathcal{O}(\log N)$ time.

However, sorting the list initially comes at a cost of $\mathcal{O}(N \log N)$. This cost might be worth paying if we expect to perform multiple search operations.

Now let M be the number of search operations we need to perform.

- Without sorting, the total time complexity is $\mathcal{O}(M \cdot N)$.
- With sorting, the total cost becomes $\mathcal{O}(N \log N + M \log N)$.

We can determine when it becomes worthwhile to pre-sort the list. We want to find the smallest M such that:

$$N \log N + M \log N < MN$$

Solving the inequality:

$$M(N - \log N) > N \log N$$

$$M \geq \left\lceil \frac{N \log N}{N - \log N} \right\rceil$$

assuming $\mathrm{N} \in \mathbb{Z} > 0$

For example let $N = 1000$:

$$M \geq \left\lceil \frac{1000 \cdot \log(1000)}{1000 - \log(1000)} \right\rceil \approx \left\lceil \frac{3000}{997} \right\rceil = 4$$

thus, if we perform at least 4 searches, it's more efficient to sort the list beforehand.

N	M	M*N	NlogN + MlogN
1000	2	2000	1000*3 + 2*3 = 3006
1000	3	3000	1000*3 + 3*3 = 3009
1000	4	4000	1000*3 + 4*3 = 3012

Figure 1: Example of time complexity trade-off.

1.1 Vertical vs Horizontal Scalability

The search problem becomes challenging when:

- N is very large (**vertical scalability** issue).
- Data items are complex and high-dimensional (**horizontal scalability** issue).

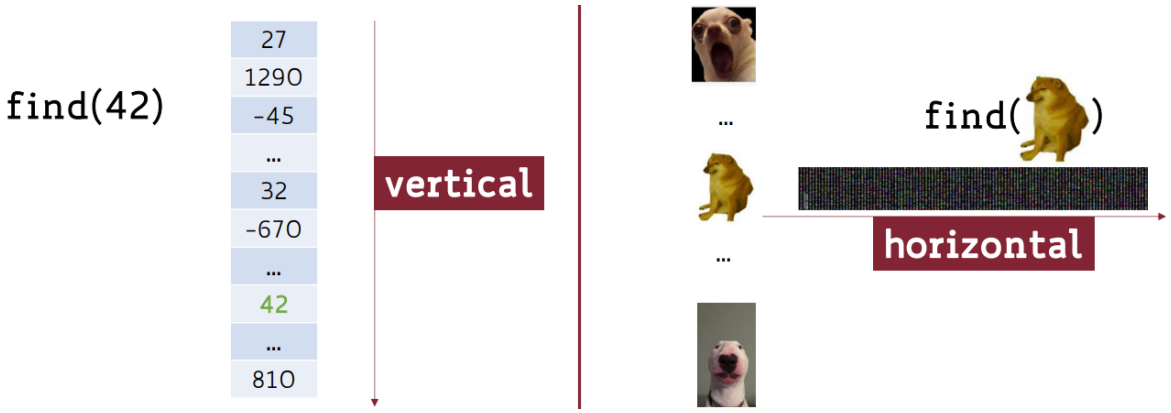


Figure 2: Vertical vs horizontal scale.

For instance, if N is so large that the dataset cannot fit into memory, we must use external sorting techniques like R-way merge sort.

Now consider a scenario where the data is not simple integers, but instead consists of images. Each image might need to be represented by a high-dimensional vector. For example, a 100×100 RGB image corresponds to a 30.000-dimensional vector of real values.

1.2 Similarity Measures

Many big data tasks require computing similarity between domain items. When the task involves searching for similar items (not just exact matches), the notion of similarity becomes critical.

- The definition of similarity is highly domain-specific.
- It is central to tasks like search, retrieval, clustering, and classification.

We often assume data lies in a d-dimensional Euclidean space, and similarity is computed using one of the following methods:

- **Euclidean Distance**: Measures the straight-line distance between two points.
- **Cosine Similarity**: Measures the cosine of the angle between two vectors.
- **Jaccard Coefficient**: Measures the similarity between finite sets.

- **...**

Choosing the right similarity measure depends on what aspect of the data we aim to emphasize and the representation of each item.

1.2.1 Metric and Metric Space

Let X be a set and let $\delta : X \times X \to [0, \infty)$ be a function. δ is called a **metric** (or **distance function**) if it satisfies the following properties for all $x, y, z \in X$:

1. **Non-negativity:** $\delta(x, y) \geq 0$
2. **Identity of indiscernibles:** $\delta(x, y) = 0$ if and only if $x = y$
3. **Symmetry:** $\delta(x, y) = \delta(y, x)$
4. **Triangle inequality:** $\delta(x, y) \leq \delta(x, z) + \delta(z, y)$

if these conditions hold, then (X, δ) is also called a **metric space**.

1.2.2 Euclidean Metric and Euclidean Space

Let $X = \mathbb{R}^d$, where d is the dimensionality of the space. For two points $x = (x_1, \ldots, x_d)$ and $y = (y_1, \ldots, y_d)$ in $\mathbb{R}^d$, the **Euclidean distance** is defined as:

$$\delta(x, y) = \sqrt{\sum_{i=1}^{d} (x_i - y_i)^2}$$

this is also referred to as the L^2 norm of the displacement vector $x - y$:

$$\|x - y\|_2 = \sqrt{(x - y) \cdot (x - y)}$$

In fact the Euclidean norm of a vector $x = (x_1, \ldots, x_d)$ measures the length of x from the origin:

$$\|x\|_2 = \sqrt{x_1^2 + \cdots + x_d^2} = \sqrt{x \cdot x}$$

The following are the cases for $d = 1$ and $d = 2$:

- **1-Dimensional Case ($d = 1$):** $x, y \in \mathbb{R}$

  $$\delta(x, y) = \|x - y\|$$

  this is the absolute value of the difference between two real numbers.

- **2-Dimensional Case ($d = 2$):** $x = (x_1, x_2)$, $y = (y_1, y_2)$

  $$\delta(x, y) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}$$

  this is directly derived from the Pythagorean theorem.

1.2.3 Minkowski Distance (L^p Norm)

The Minkowski distance generalizes the Euclidean distance. For $x, y \in \mathbb{R}^d$ and $p \geq 1$, the L^p norm is defined as:

$$\delta_p(x, y) = \left(\sum_{i=1}^{d} \|x_i - y_i\|^p \right)^{1/p}$$

The following are some special cases:

- $p = 1$ **(Manhattan Distance):**

$$\delta_1(x, y) = \sum_{i=1}^{d} \|x_i - y_i\|$$

- $p = 2$ **(Euclidean Distance):**

$$\delta_2(x, y) = \left(\sum_{i=1}^{d} \|x_i - y_i\|^2 \right)^{1/2}$$

- $p \to \infty$ **(Chebyshev Distance):**

$$\delta_\infty(x, y) = \max_{i=1,\ldots,d} \|x_i - y_i\|$$

1.2.4 Cosine Similarity

Cosine similarity measures the cosine of the angle between two non-zero vectors in an inner product space. It is defined as:

$$\cos(\theta) = \frac{x \cdot y}{\|x\| \, \|y\|}$$

where θ is the angle between vectors x and y, and $x \cdot y$ is their dot product.

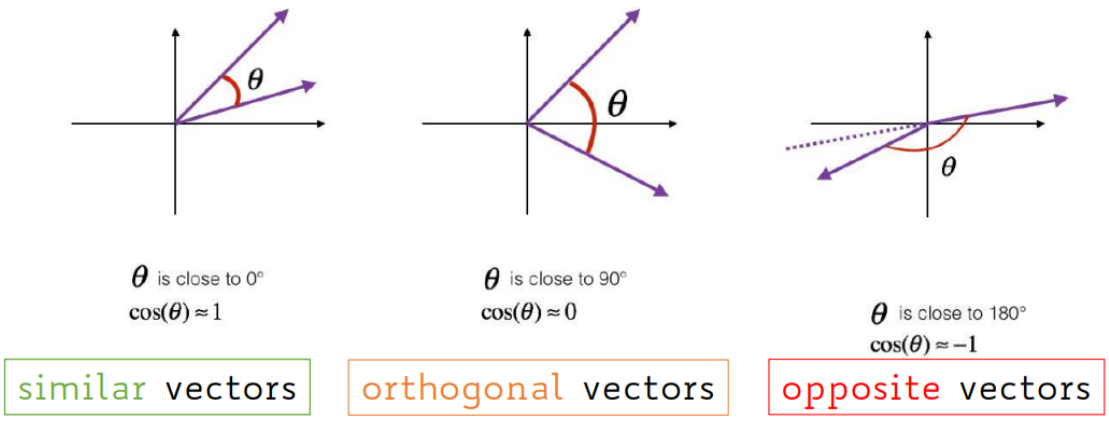


Figure 3: Cosine similarity cases.

The following are some cases:

- **2D Case:**
  if: $x = (\|x\| \cos\alpha, \|x\| \sin\alpha), \quad y = (\|y\| \cos\beta, \|y\| \sin\beta)$
  then: $x \cdot y = \|x\|\|y\|(\cos\alpha\cos\beta + \sin\alpha\sin\beta) = \|x\|\|y\|\cos(\theta)$
  notice that $\cos\theta = x \cdot y / \|x\|\|y\|$

- **General Case:**
  in $\mathbb{R}^d$, if two vectors x and y are not collinear, they span a 2D plane within $\mathbb{R}^d$, and the cosine similarity is computed in the same way.

1.2.5 Jaccard Similarity and Distance

The Jaccard Index (Coefficient) measures similarity between two finite sets A and B:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

there is also the special case where $A = B = \emptyset$, then $J(A,B) = 1$.

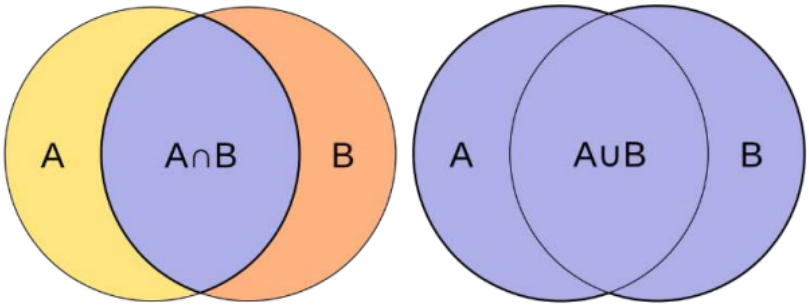


Figure 4: Jaccard coefficient interpretation.

The Jaccard distance is a metric, is complementary to the Jaccard coefficient and is defined as:

$$\delta_J(A,B) = 1 - J(A,B) = \frac{|A \cup B| - |A \cap B|}{|A \cup B|}$$

2 The Curse of Dimensionality

Each similarity metric or distance discussed in the previous chapter can be suitable for a specific task and domain. For example, clustering is one such task where the choice of similarity measure plays a crucial role.

2.1 Clustering

Clustering is a procedure to group a set of objects into classes of similar objects, and is a standard problem in many big data applications. Clustering also belongs to the unsupervised learning techniques and is used to explore data by identifying patterns or grouping similar data points together.

A formal **definition for clustering** is: Given a set of data points and a notion of distance between those, clustering is the task of grouping the data points into some number of clusters so that the members of a cluster are 'close' to each other and the members of different clusters are dissimilar to each other.

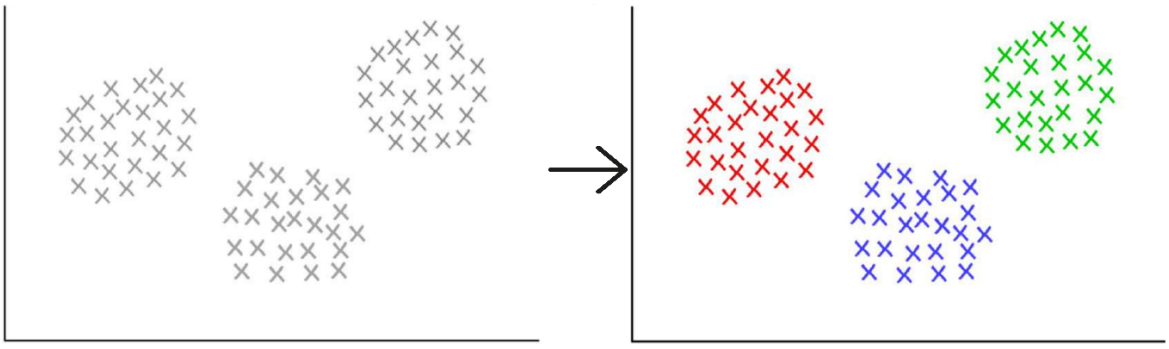


Figure 5: Clustering ideally.

Clustering also presents several challenges, including:

- **Object representation**: data points may reside in very high-dimensional spaces, making their representation and manipulation more complex.
- **Notion of similarity**: choosing an appropriate similarity or distance measure between objects is crucial.
- **Number of clusters**: the number of clusters can either be predefined or determined in a data-driven manner, depending on the context and algorithm used.

Additionally, even if clustering may seem easy in two dimensions or with a small number of data points, it becomes significantly more challenging in real-world applications, where data often involves tens, hundreds, or even thousands of dimensions. In such high-dimensional spaces, distances between points tend to

become uniformly large; in other words, almost all pairs of points are approximately the same (large) distance apart.

2.1.1 Example: Text Document Clustering

Suppose we want to group together documents on the same topic. We will soon be faced with two problems:

- How to represent documents,
- How to measure similarity between documents.

There are different ways of representing documents (in the space of words):

- As a set of words (disregarding the order and multiplicity)
- As a bag-of-words (a multiset disregarding the order yet keeping multiplicity)
- As a bag-of-n-grams (the more general case of bag-of-words)
- More advanced representations derived from Neural Language Models (e.g., word2vec, BERT)

Obviously, the choice of document representation affects the similarity measure.

The bag-of-words model is just a preliminary step for more complex document representations.

The bag-of-words can be seen as a set of (key, value) pairs, where the key is a word and the value is the number of its occurrences in the document.

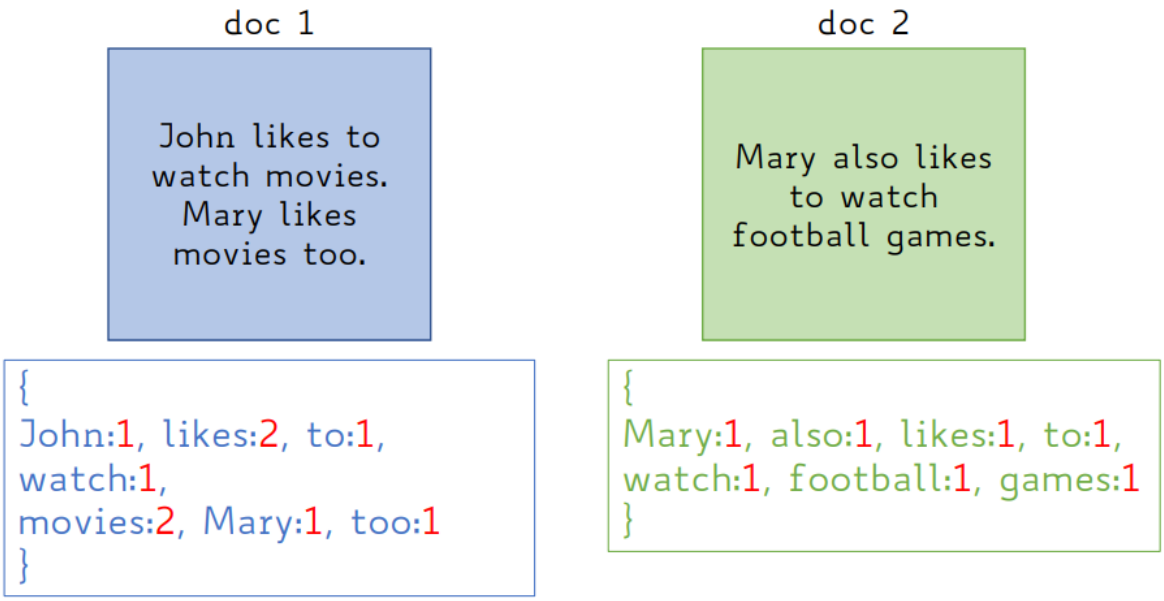


Figure 6: Bag-of-words as set of (key, value) pairs.

In practice, it can be represented using a vocabulary, which is a fixed, alphabetically sorted list of all possible words, and a weighting scheme. In this case, each document is mapped to a vector where the value at position i indicates the number of times the i-th word in the vocabulary appears in the document. This representation allows the use of a single vocabulary across multiple documents and assigns a zero when a word in the vocabulary simply does not appear. In a formal perspective, we have:

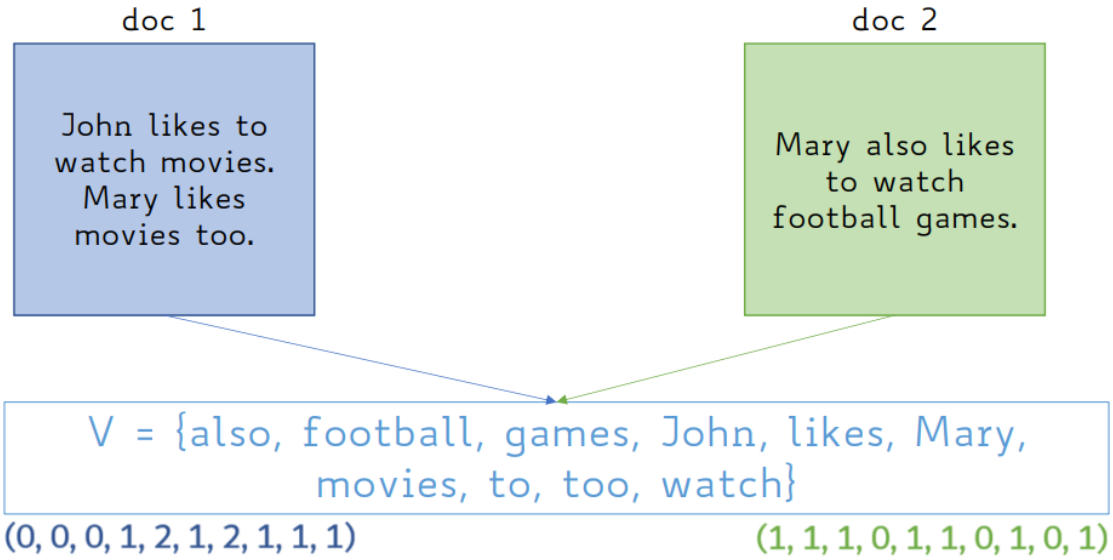


Figure 7: Bag-of-words with vocabulary and term-frequency weighting scheme.

- $D = \{d_1, ..., d_N\}$ a collection of N documents
- $V = \{w_1, ..., w_{|V|}\}$ a **vocabulary** of $|V|$ words extracted from D
- $\hat{d_i} = (f(w_1, i), ..., f(w_{|V|}, i))$ a $|V|$-dimensional vector representing d_i
- $f : V \times D \rightarrow \mathbb{R}$ a function that maps each word of a document to a real value (**weighting scheme**)

Consequently, the weighting scheme does not always involve counting word occurrences, as in the case shown in figure 7. For example, in the **one-hot (or binary) weighting scheme**, we simply set $f(w_j, i) = 1$ if w_j appears in d_i, and 0 otherwise.

There is, therefore, the case of the **term-frequency weighting scheme**, which is the one shown in figure 7. In which $f(w_j, i)$ becomes $tf(w_j, i)$ to distinguish it from the previous case.

But there is also the **TF-IDF weighting scheme**, where $f(w_j, i)$ becomes:

$$f(w_j, i) = tf(w_j, i) \cdot idf(w_j)$$

where $idf(w_j) = log(\frac{N}{n_j})$ and n_j is the number of documents in D containing the word w_j. Moreover, $idf(w_j)$ often becomes $log(\frac{N+1}{n_j+1})$ in order to avoid divisions by zero.

The bag-of-words (BoW) model has two main limitations: it results in high-dimensional and sparse representations, and it lacks both word order and semantic meaning, as it treats text as a simple collection of unigrams. These issues can be mitigated by using n-grams to capture local context or, more effectively, by adopting Neural Language Models such as word2vec or BERT, which incorporate both syntax and semantics.

In figure 8, an example of using the bag-of-n-grams model with $n = 2$ is shown, using the same documents as in the previous example.

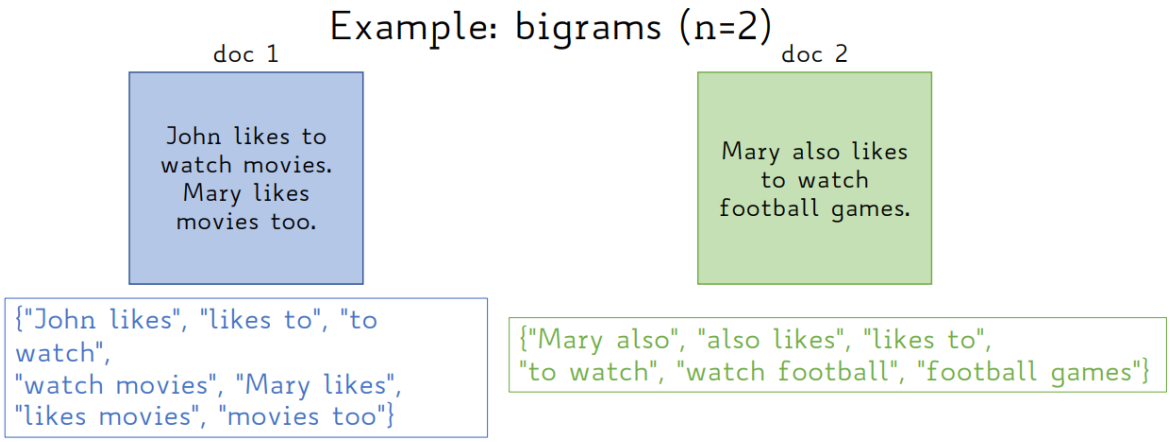


Figure 8: Bag-of-n-grams with $n = 2$.

2.2 Dimensionality

Different document representations allow for different similarity measures. For instance, sets of words are typically compared using the Jaccard coefficient, one-hot bag-of-words representations can use Euclidean distance, while tf or tf-idf weighted models are well-suited for cosine similarity.

In many domains, such as text, images, or audio, data is often represented in very high-dimensional spaces. For example, in the word space, the vocabulary size can be extremely large, yet most documents only contain a small subset of those words. This leads to sparse representations where only a few dimensions are non-zero and data points tend to be more dissimilar from each other.

This happens because in Euclidean space, two points are considered far apart if they differ significantly in at least one dimension, but as the number of dimensions increases, the likelihood of this happening grows, leading to what is known as the **Curse of Dimensionality**.

To better understand this, let us define H as a hypercube (a generalization of a cube to d dimensions), where d is the number of dimensions of the space, and let N be the number of data points uniformly distributed within H. Given a random point p in H, we define l as the edge length of the hypercube h inscribed in H that contains

p along with its k nearest points. We consider edge points whose distance from p is at most $\frac{l}{2}\sqrt{d}$.

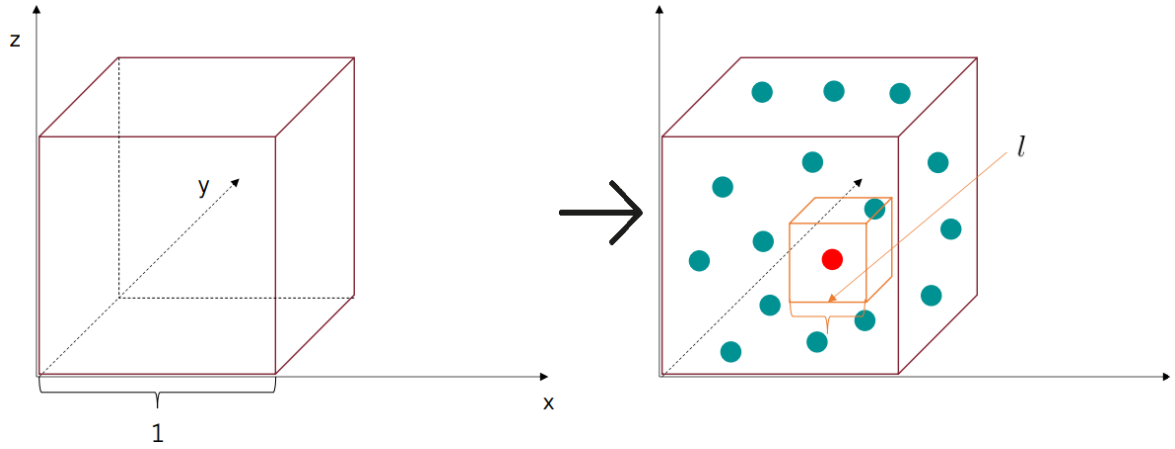


Figure 9: Hypercube with uniformly distributed points.

The same question can also be formulated in terms of the radius l of an inscribed hypersphere. Therefore, let $V_h = l^d$ be the volume of the hypercube h; it should roughly contain $\frac{k}{N}$ points since those are randomly distributed.

$$l^d \approx \frac{k}{N} \iff l \approx (\frac{k}{N})^{(1/d)}$$

d	l
1	0.01
2	0.1
3	0.215
...	...
10	0.631
...	...
1000	0.995

$$N = 1,000; k = 10 \quad l \approx \left(\frac{10}{1000}\right)^{1/d} = \left(\frac{1}{100}\right)^{1/d}$$

When d is equal 10 the length of the edge of the inscribed hypercube is already about 63% of the largest hypercube

When d is equal 1,000 there is basically no difference between the two hypercubes!

Figure 10: The curse of dimensionality in practice.

In high-dimensional spaces, most data points tend to lie near the edges, making even the nearest neighbors relatively far apart. This leads to a phenomenon called distance concentration, where distances between points become almost indistinguishable, making it difficult to identify clusters or meaningful neighbors. This is related to the 'edge' effect in high-dimensional spaces, which refers to the fact that, as the number of dimensions increases, the probability of selecting a point not near the boundary of the space rapidly decreases. In other words, most points end up being close to the edge.

Let ϵ define the edge of our space. The probability that a point is not near the edge is given by $(1-2\epsilon)^d$, assuming that each dimension is independent. Naturally, as the number of dimensions increases:

$$\lim_{d\to\infty}(1-2\epsilon)^d = 0$$

If data were truly uniformly distributed in a high-dimensional space, there wouldn't be much we could do. Fortunately, real-world data often exhibit underlying patterns (they are not completely random) and typically lie on structures of much lower intrinsic dimensionality.

The **Manifold Hypothesis** suggests that high-dimensional data often reside on low-dimensional manifolds embedded within the high-dimensional space. This insight is the foundation for dimensionality reduction techniques.

Concrete example: Images of human faces contain millions of pixels (and thus seem to exist in a space with millions of dimensions), but in reality, the variations between one face and another (e.g., expressions, angle, lighting) occur along a much smaller number of dimensions. This hidden space is the manifold. So, a manifold is a "surface" (of any dimension) that represents the true underlying structure of data in high-dimensional spaces.

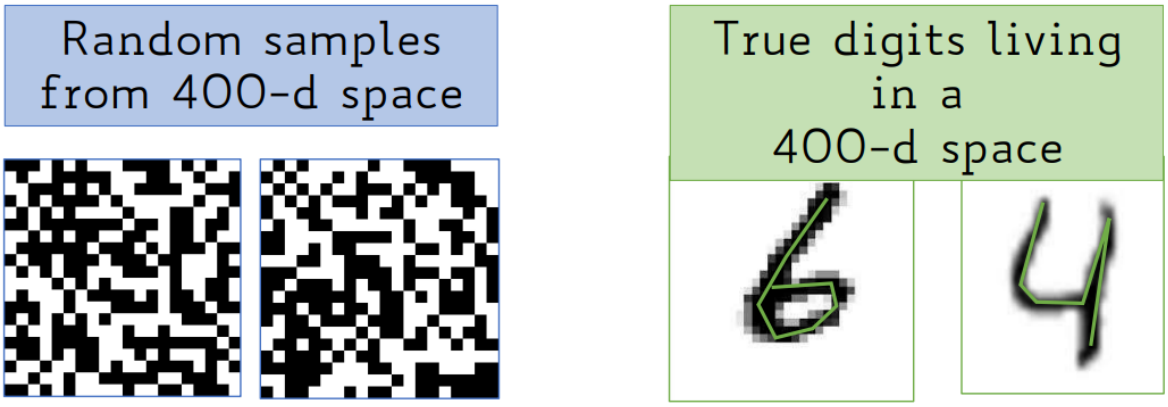


Figure 11: Modeled vs true dimensionality.

3 Clustering

As discussed in the previous chapter, clustering is an unsupervised learning method that groups similar data objects based on their representation and the chosen similarity measure. The problem becomes increasingly challenging in high-dimensional spaces due to the so-called curse of dimensionality, and a crucial aspect is also determining the appropriate number of clusters.

3.1 Clustering Algorithms

Clustering algorithms are typically divided into five categories:

- **Partitioning-based**
- **Hierarchical-based**
- **Density-based**
- **Grid-based**
- **Model-based**

figure 13 shows some examples for each type of algorithm.

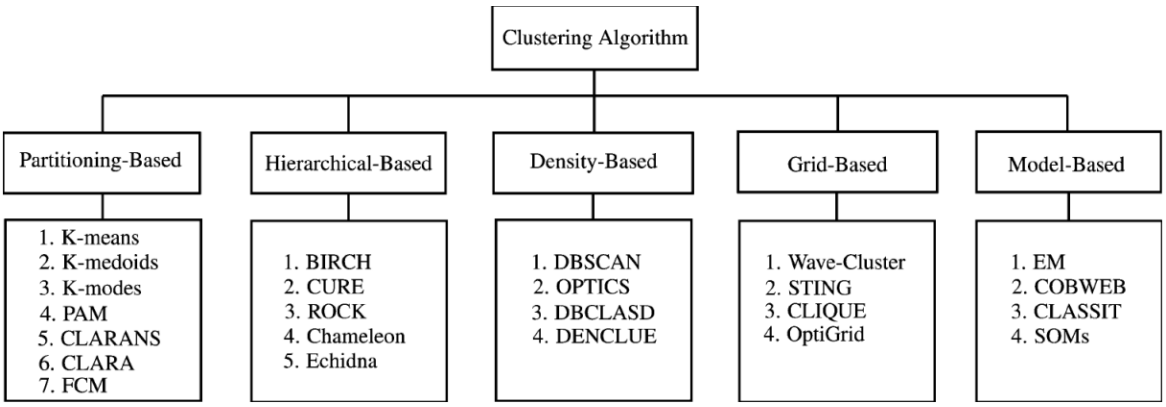


Figure 12: Taxonomy of clustering algorithms.

3.1.1 Partitioning

Partitioning (or hard) clustering takes as input a set of N data points and a desired number of clusters K, with $K < N$, and outputs a division of the data into K non-overlapping clusters. The objective is to find the partition that best optimizes a given criterion.

What happens in practice is that, given N data points and a specified number of clusters K, we use an N-dimensional assignment vector $\mathbf{C}$, where $C[i] \in \{0, \ldots, K-1\}$ indicates the cluster to which the i-th data point is assigned.

In general, the number of possible clustering outputs is approximately K^N, actually slightly fewer in practice, since we usually require that each cluster contains at least one data point. So, for example, if $K = 2$, the assignment vector C cannot consist entirely of 0s or 1s ([0,0,...,0] or [1,1,...,1]).

Partitioning-based clustering is **NP-hard** (non-deterministic polynomial-time hard) because the number of possible non-empty K-way partitions of N elements (given by Stirling numbers) grows rapidly, on the order of K^N. This makes it infeasible to enumerate all possible partitions to find the global optimum. Consequently, practical approaches rely on heuristic algorithms such as K-means, K-medoids, or K-means++.

Flat hard clustering is a general framework that defines:

- $x_1, \ldots, x_N$ as the set of N input data points
- $C_1, \ldots, C_K$ as the set of K output clusters
- C_k as the generic k-th cluster
- θ_k as the representative of cluster C_k

where the representative of a cluster is an element that summarizes, in some way, for instance through an average, all the data points assigned to that cluster. The term flat refers to the fact that the clusters are not connected by hierarchical relationships, while hard means that each data point is assigned to exactly one cluster, making the clusters disjoint.

In clustering, the *objective function* $L(A, \Theta)$ quantifies the total cost (or error) of assigning data points to clusters. Formally, it is defined as:

$$L(A, \Theta) = \sum_{n=1}^{N} \sum_{k=1}^{K} \alpha_{n,k}\, \delta(x_n, \theta_k)$$

where:

- A is an $N \times K$ assignment matrix with entries $\alpha_{n,k}$ such that $\alpha_{n,k} = 1$ if the point x_n is assigned to cluster C_k, and 0 otherwise. Each data point is assigned to exactly one cluster, so for every n, there is exactly one k with $\alpha_{n,k} = 1$,
- $\Theta = \{\theta_1, \ldots, \theta_K\}$ is the set of cluster representatives,
- $\delta(x_n, \theta_k)$ is a distance function measuring the dissimilarity between x_n and θ_k.

The goal of clustering is to find the optimal assignment A^* and representatives Θ^* minimizing the objective function:

$$(A^*, \Theta^*) = \arg\min_{A,\Theta} L(A, \Theta).$$

However, finding the exact global optimum is computationally infeasible due to the exponential size of the search space $S(K, N)$, which grows roughly as $\mathcal{O}(K^N)$.

Additionally, the problem is *non-convex* because of the discrete nature of the assignment matrix A, leading to multiple local minima.

The **Lloyd-Forgy algorithm** provides an iterative approximation to clustering, since NP-hardness and non-convexity prevent finding the exact global optimum. It alternates between an **assignment step** and an **update step**, but it may get stuck in a local optimum or saddle point and does not guarantee a globally optimal solution.

The assignment step minimizes the objective function L with respect to the assignment matrix A, while keeping the cluster representatives Θ fixed. Since A is discrete, gradients cannot be used. Intuitively, this step assigns each data point to the closest cluster representative according to the distance function δ, thereby minimizing the total distance from points to their assigned representatives.

$$\alpha_{n,k} = \begin{cases} 1 & \text{if } \delta(x_n, \theta_k) = \min\limits_{1 \le j \le K} \delta(x_n, \theta_j) \quad (\text{if } \theta_k \text{ is the closest representative to } x_n) \\ 0 & \text{otherwise} \end{cases}$$

The update step consists of minimizing the objective function L with respect to the cluster representatives Θ, while keeping the assignment matrix A fixed. Since Θ consists of continuous variables, we can compute the gradient of L with respect to Θ, set it to zero, and solve the resulting system to update the cluster representatives.

$$\nabla L(\Theta; A) = \left(\frac{\partial L(\Theta; A)}{\partial \theta_1}, \ldots, \frac{\partial L(\Theta; A)}{\partial \theta_K} \right)$$

where

$$\frac{\partial L(\Theta; A)}{\partial \theta_j} = \frac{\partial L(\theta_1, \ldots, \theta_K; A)}{\partial \theta_j}$$

is the partial derivative with respect to the generic cluster representative θ_j. We can say that $\nabla L(\Theta; A) = 0 \iff \frac{\partial L(\theta_1, \ldots, \theta_K; A)}{\partial \theta_j} = 0 \quad \forall j \in \{1, \ldots, K\}$.

To simplify the notation, we denote $\frac{\partial L(\theta_1, \ldots, \theta_K; A)}{\partial \theta_j}$ as $\frac{\partial L}{\partial \theta_j}$, thus:

$$\frac{\partial L}{\partial \theta_j} = \frac{\partial}{\partial \theta_j} \left[\sum_{n=1}^{N} \sum_{k=1}^{K} \alpha_{n,k} \delta(x_n, \theta_k) \right]$$

When computing the partial derivative with respect to θ_j, all other terms θ_k in the summation are treated as constants. This implies that the derivative simply becomes:

$$\frac{\partial L}{\partial \theta_j} = \frac{\partial}{\partial \theta_j} \left[\sum_{n=1}^{N} \alpha_{n,j} \delta(x_n, \theta_j) \right] = 0$$

and is then solved for each θ_j, depending on the specific distance function δ.

3.1.2 K-Means

K-means is a special case of partitioning-based clustering where each cluster is represented by its centroid, which consists of the mean of the points assigned to it. The algorithm iteratively reassigns points to the closest centroid and updates the centroids, aiming to minimize the total within-cluster sum of squared distances (SSD). So the representative:

$$\theta_k = \frac{\sum_{n=1}^{N} \alpha_{n,k} x_n}{\sum_{n=1}^{N} \alpha_{n,k}} = \mu_k = \frac{1}{|C_k|} \sum_{n \in C_k} x_n \quad \text{where} \quad |C_k| = \sum_{n=1}^{N} \alpha_{n,k}$$

and where $\{x_1, \ldots, x_N\}$ is the set of N data points, and $\{C_1, \ldots, C_K\}$ is the set of K output clusters.

The objective function becomes:

$$L(A, \Theta) = \sum_{n=1}^{N} \sum_{k=1}^{K} \alpha_{n,k} \left(\|x_n - \theta_k\|_2 \right)^2$$

where δ becomes a function that computes the Euclidean distance (seen in chapter 1) between x_n and the centroid θ_k:

$$\delta(x_n, \theta_k) = (\|x_n - \theta_k\|_2)^2 = [\sqrt{(x_n - \theta_k)^2}]^2 = (x_n - \theta_k)^2$$

thus, ultimately the objective function becomes:

$$L(A, \Theta) = \sum_{n=1}^{N} \sum_{k=1}^{K} \alpha_{n,k} (x_n - \theta_k)^2$$

In the assignment step of the K-means algorithm, the objective function L is minimized with respect to the assignment matrix A by fixing the set of centroids Θ. Intuitively, each data point x_n is assigned to the centroid θ_k that minimizes the squared Euclidean distance, resulting in:

$$\alpha_{n,k} = \begin{cases} 1 & \text{if } \|x_n - \theta_k\|^2 = \min\limits_{1 \le j \le K} \|x_n - \theta_j\|^2 \\ 0 & \text{otherwise} \end{cases}$$

In the update step, the objective function is minimized with respect to the set of centroids Θ, by fixing the assignment matrix A. This corresponds to solving the following optimization problem:

$$\Theta^* = \arg\min_{\Theta} \sum_{n=1}^{N} \sum_{k=1}^{K} \alpha_{n,k} \left(x_n - \theta_k \right)^2$$

To find the optimal Θ, we compute the gradient of L with respect to Θ, set it equal to zero, and solve for each θ_k.

The partial derivative becomes:

$$\frac{\partial L}{\partial \theta_k} = \frac{\partial}{\partial \theta_k}[\sum_{n=1}^{N} \alpha_{n,k}(x_n - \theta_k)^2] = \sum_{n=1}^{N} -2\alpha_{n,k}(x_n, \theta_k)$$

and we want to find each θ_k^* such that:

$$\sum_{n=1}^{N} -2\alpha_{n,k}(x_n, \theta_k^*) = 0$$

setting the partial derivative equal to zero, we obtain:

$$\sum_{n=1}^{N} -2\alpha_{n,k}(x_n - \theta_k^*) = 0 \iff 2\sum_{n=1}^{N} \alpha_{n,k}\theta_k^* = 2\sum_{n=1}^{N} \alpha_{n,k}x_n \iff \theta_k^* \sum_{n=1}^{N} \alpha_{n,k} = \sum_{n=1}^{N} \alpha_{n,k}x_n$$

since θ_k^* does not depend on n, it can be factored out of the summation. A further step is to derive the formula for θ_k^*, which turns out to be the first formula of this subchapter, μ_k:

$$\theta_k^* = \frac{\sum_{n=1}^{N} \alpha_{n,k}x_n}{\sum_{n=1}^{N} \alpha_{n,k}} = \mu_k = \frac{1}{|C_k|} \sum_{n \in C_k} x_n$$

thus, in the update step of the K-means algorithm, the new centroid θ_k^* of cluster C_k is computed using the formula for μ_k.

The **Lloyd-Forgy** is the standard approach for implementing a **K-means algorithm**. It follows five steps:

1. Specify the number of output clusters K
2. Select K observations at random from the N data points as the initial cluster centroids
3. **Assignment step**: Assign each observation to the closest centroid based on the chosen distance measure
4. **Update step**: For each of the K clusters, update the centroid by computing the new mean of all the data points assigned to that cluster
5. Iteratively repeat steps 3–4 until a stopping criterion is met

This stopping criterion can vary, but common options include: setting a fixed number of iterations, stopping when cluster assignments no longer change (beyond a certain threshold), or when the centroids themselves stabilize (within a certain threshold).

This algorithm is guaranteed to converge to a fixed point where cluster assignments no longer change. This happens because each iteration either improves or maintains the objective function, and the algorithm is a specific case of the more general **Expectation-Maximization** (EM) framework, which is known to converge, though

not necessarily to a global optimum. The assignment step corresponds to the E-step (Expectation), where each data point is assigned to the closest centroid, and the update step corresponds to the M-step (Maximization), where centroids are recomputed to minimize the within-cluster sum of squared distances (SSD). Both steps guarantee a monotonic decrease in the total SSD.

The computational complexity of this Lloyd-Forgy algorithm can be analyzed as follows:

- Computing the distance between two d-dimensional data points takes $\mathcal{O}(d)$.
- **Assignment step**: Assigning each of the N data points to the nearest of K centroids requires $\mathcal{O}(KN)$ distance computations, each costing $\mathcal{O}(d)$, resulting in a total complexity of $\mathcal{O}(KNd)$.
- **Update step**: Updating the centroids involves computing the mean of all data points in each cluster. Since each of the N points contributes to one cluster and each update is $\mathcal{O}(d)$, this step takes $\mathcal{O}(Nd)$.
- **Overall complexity**: Assuming the algorithm runs for R iterations, the total time complexity is $\mathcal{O}(RKNd)$

The convergence rate and clustering quality in K-means depend heavily on the initial centroids. Random methods such as the Forgy method, which randomly selects K data points as initial means, and the Random Partition method, which randomly assigns clusters to each observation, often lead the algorithm to sub-optimal solutions. A common way to mitigate this issue is to run the Lloyd-Forgy algorithm multiple times with different random initializations.

An interesting method to carefully select initial centroids is the **K-means++** proposed by Arthur and Vassilvitskii in 2007. The key idea is to choose the initial centroids in a way that they are well spread out, selecting each new centroid as the data point farthest from the ones already chosen:

1. Choose one centroid uniformly at random from the set of data points.
2. For each data point x, compute $D(x)$, the distance between x and the nearest centroid already chosen.
3. Select a new data point as the next centroid with probability proportional to $D(x)^2$.
4. Repeat steps 2 and 3 until K centroids have been chosen, then run the Lloyd-Forgy algorithm.

Compared to the standard "vanilla" K-means, K-means++ improves reliability by providing a theoretical guarantee: the resulting clusters are at most $O(\log K)$ times

worse than the optimal solution. This makes K-means++ both more stable and more accurate in practice.

In clustering problems, the number of clusters K may sometimes be predetermined; however, in most cases, selecting K is part of the problem itself. The goal is therefore to strike a balance between having too many or too few clusters, aiming to achieve the best possible trade-off between total benefit and total cost.

The **total benefit** is defined as the sum of individual benefits b_i, where each b_i measures the similarity of a data point x_i to its assigned centroid. Notably, there always exists a clustering where the total benefit $B = N$ where N is the number of data points (by assigning each point to its own cluster). The **total cost** is defined by assigning a cost p to each cluster, so that a clustering with K clusters has a total cost $P = Kp$. The **overall value** V of a clustering is then given by the total benefit minus the total cost, $V = B - P$. The goal is to find the number of clusters K that maximizes V.

The **Elbow method** is an empirical approach to determine the optimal number of clusters K by balancing total benefit and total cost. It involves testing multiple values of K and observing the change in the sum of squared distances (SSD), which typically decreases sharply up to a certain point before leveling off.

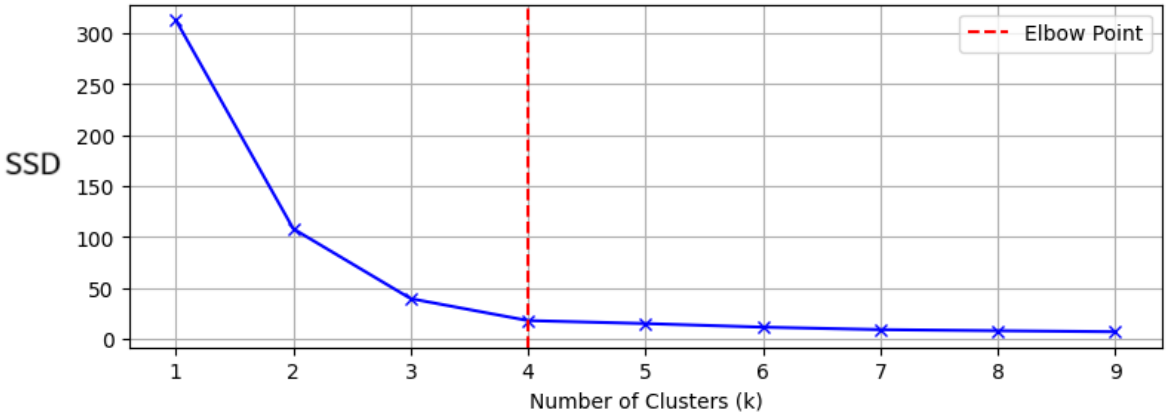


Figure 13: Elbow method for optimal K.

Until now, we focused on Euclidean distance, but the hard clustering framework can be applied using other distance measures as well. Some distances, like cosine or correlation distance just resemble the Euclidean distance, so centroids still minimize the objective; others, such as Manhattan distance (L_1-norm), require different minimizers like the median, as used in K-medians.

A further approach is **K-medoids**, which is similar to K-means but selects actual data points (medoids) as cluster centers, and works with any arbitrary distance measure. A medoid is the closest object to any other point in the cluster. The **Partitioning Around Medoids** (PAM) algorithm, introduced by Kaufman and Rousseeuw in 1987, is a popular greedy approach that implements K-medoids.

K-medoids is more robust to outliers but computationally expensive, with complexity $O(K(N-K)^2)$.

The **Bradley-Fayyad-Reina** (BFR) algorithm is another variant of K-means specifically designed for large-scale datasets, particularly in high-dimensional Euclidean spaces. It assumes that clusters are normally distributed around their centroids and that data dimensions are independent, which helps it efficiently scale and perform clustering under these conditions.

3.2 Measures of Clustering Quality

The measures that determine clustering quality are obtained through internal evaluation and external evaluation.

Internal evaluation assesses the quality of a clustering using the data that was clustered itself. The idea is that a good clustering should show high similarity within clusters (intra-cluster) and low similarity between different clusters (inter-cluster). This kind of evaluation depends on how the data is represented and the similarity measure used.

The **Davies-Bouldin Index** is an internal evaluation metric. Lower values of this index indicate better clustering quality, and it is defined as:

$$DB = \frac{1}{K}\sum_{i=1}^{K}\max_{j\neq i}\left(\frac{\sigma_i+\sigma_j}{\delta(\mu_i,\mu_j)}\right)$$

where:

- K is the number of clusters;
- μ_k is the centroid of cluster C_k;
- σ_k is the average distance of all elements in cluster C_k from its centroid μ_k;
- $\delta(\mu_i,\mu_j)$ is the distance between the centroids of clusters C_i and C_j.

The **Dunn Index** is another internal evaluation metric where a higher value indicates better clustering quality. It is defined as:

$$D = \min_{1\leq i<j\leq K}\left(\frac{\delta(C_i,C_j)}{\max_{1\leq k\leq K}\delta'(C_k)}\right)$$

where:

- $\delta(C_i,C_j)$ is the distance between clusters C_i and C_j, which is the distance between their centroids;
- $\delta'(C_k)$ is the intra-cluster distance of cluster C_k, which is the maximum distance between any two objects in the cluster.

The **Silhouette Coefficient** is a further internal evaluation metric that measures how similar an object is to its own cluster compared to other clusters. It is defined as follows:

$$a(i) = \frac{1}{|C_i| - 1} \sum_{\substack{j \in C_i \\ j \neq i}} \delta(i, j)$$

where $a(i)$ is the average distance between point i and all other points in the same cluster C_i;

$$b(i) = \min_{k \neq i} \left(\frac{1}{|C_k|} \sum_{j \in C_k} \delta(i, j) \right)$$

where $b(i)$ is the smallest average distance from point i to all points in any other cluster $C_k \neq C_i$. The silhouette coefficient $s(i)$ for point i is then defined as:

$$s(i) = \begin{cases} 1 - \frac{a(i)}{b(i)} & \text{if } a(i) < b(i) \\ 0 & \text{if } a(i) = b(i) \\ \frac{b(i)}{a(i)} - 1 & \text{if } a(i) > b(i) \end{cases}$$

A higher silhouette coefficient indicates better clustering quality.

External evaluation assesses clustering quality using external, pre-labeled (gold standard) data that was not involved in the clustering process. It measures how well the clustering algorithm uncovers the true underlying structure, but it requires labeled data, which is often difficult and costly to obtain as it typically comes from human experts.

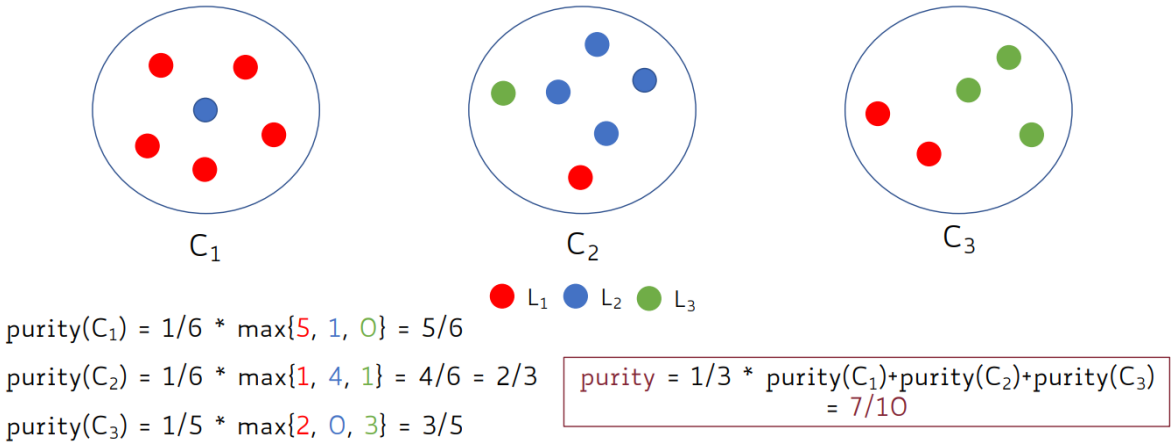


Figure 14: External evaluation: purity example.

Purity is a method that falls under **external evaluation**. Let $C_1, \ldots, C_K$ be the set of K clusters, and $L_1, \ldots, L_J$ the set of J true labels. Define:

- $n_{i,j}$: the number of items in cluster C_i with label L_j
- $n_i = \sum_{j=1}^{J} n_{i,j}$: the total number of items in cluster C_i

then, the purity of a single cluster is:

$$\text{purity}(C_i) = \frac{1}{n_i} \max_{j \in \{1,\dots,J\}} n_{i,j}$$

and the overall purity of the clustering is:

$$\text{purity} = \frac{1}{K} \sum_{i=1}^{K} \text{purity}(C_i)$$

Note that purity is biased, since it reaches its maximum value when the number of clusters equals the number of data points.

The **Rand Index** is an external evaluation metric that measures the level of agreement between a clustering result and a ground truth classification. It is computed by considering all (prediction, ground truth) pairs, and counting the ones that are assigned in the same or different clusters.

$$\text{Rand Index} = \frac{TP + TN}{TP + TN + FP + FN}$$

where:

- TP is the number of true positive pairs (pairs in the same cluster in both predicted and ground truth),
- TN is the number of true negative pairs (pairs in different clusters in both predicted and ground truth),
- FP is the number of false positive pairs (pairs in the same cluster in the predicted clustering but not in the ground truth),
- FN is the number of false negative pairs (pairs in different clusters in the predicted clustering but in the same cluster in the ground truth).

n. of pairs	Same Cluster in Clustering	Different Clusters in Clustering
Same Cluster in Ground-Truth	TRUE POSITIVES (TP)	FALSE NEGATIVES (FN)
Different Clusters in Ground-Truth	FALSE POSITIVES (FP)	TRUE NEGATIVES (TN)

Figure 15: Confusion matrix.

Precision, **recall**, and the **F-measure** are further external evaluation metrics defined as follows:

$$\text{Precision (P)} = \frac{TP}{TP + FP}$$
$$\text{Recall (R)} = \frac{TP}{TP + FN}$$
$$F_\beta = \frac{(\beta^2 + 1) \cdot P \cdot R}{\beta^2 \cdot P + R}$$

where the parameter β allows weighting recall more heavily than precision when $\beta > 1$, and vice versa when $\beta < 1$. The most common case is $\beta = 1$, known as the F_1 score.

$$F_1 = \frac{2 \cdot P \cdot R}{P + R}$$

There are many other external evaluation metrics used to assess clustering quality against labeled data, including the Jaccard index, Dice index, Fowlkes-Mallows index, and Mutual Information, among others.

4 Dimensionality Reduction

As already discussed, the curse of dimensionality refers to the phenomenon where, as the number of dimensions increases, the data becomes sparse, and each region of the feature space contains fewer examples, if the total number of examples remains constant. In other words, to maintain the same data density across dimensions, the number of examples must grow exponentially with dimensionality.

There are three common approaches to dealing with high-dimensional data:

- **Feature Engineering**: Using domain-specific knowledge to design and select relevant features.
- **Making Assumptions**: Introducing prior assumptions or constraints to simplify the learning problem.
- **Reduce Dimensionality**: Creating a new smaller, set of variables that capture most of the relevant information in the original dataset.

Assumptions to deal with high dimensionality can include:

- **Independence**: each dimension is treated separately, for example by counting frequencies per feature.
- **Smoothness**: information is propagated to neighboring regions in the feature space, assuming nearby points behave similarly.
- **Symmetry**: the model is invariant to the order or permutation of dimensions.

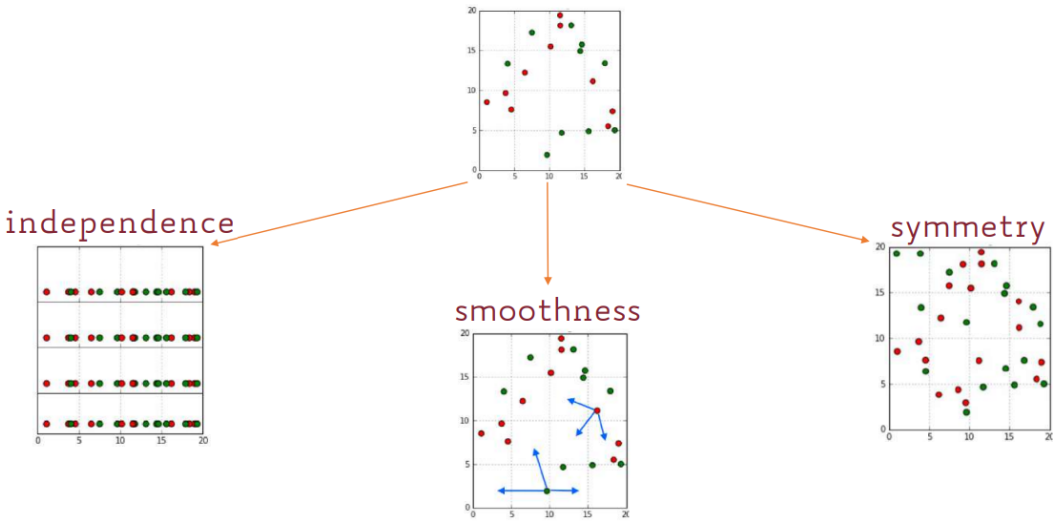


Figure 16: Assumptions to deal with high dimensionality.

Dimensionality reduction is a technique used to uncover the meaningful underlying dimensions in data by representing it with fewer features. The goal is to retain as

much of the original data structure (structure as variance) as possible, and preserving the aspects that are important for distinguishing or separating the data.

There are two main approaches to dimensionality reduction:

- **Feature Selection:** This method involves selecting a subset of the original dimensions that are most informative or predictive. For instance, techniques like information gain can be used to identify the most relevant features.
- **Feature Extraction:** In this approach, a new set of $k < d$ dimensions is constructed, typically as a linear combination of the original features. These new features, $e_1, e_2, \ldots, e_k$, are defined such that:

$$e_i = f(x_1, x_2, \ldots, x_d)$$

4.1 Principal Component Analysis

Principal Component Analysis (PCA) is a dimensionality reduction technique based on feature extraction. It defines a set of principal components as follows:

- The first principal component corresponds to the direction of maximum variance in the data.
- The second principal component is orthogonal (perpendicular) to the first and captures the maximum variance remaining in the orthogonal subspace.
- This process continues iteratively until all d components are defined, each being orthogonal to the previous ones.

The top $k < d$ principal components, which capture the most variance, are selected as the new dimensions of the transformed dataset.

In figure 17 the top two principal components, PC1 and PC2, represent the directions of highest variance in the data. Once identified, every data point is re-expressed (projected) in terms of these new axes, effectively changing its coordinates to align with the new reduced-dimensional space.

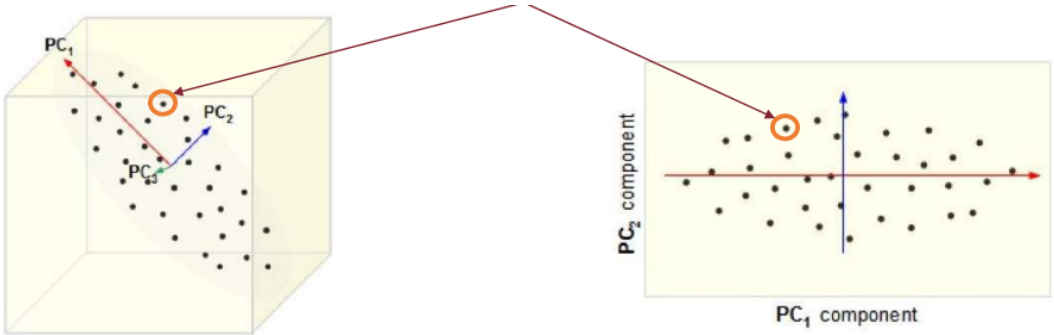


Figure 17: PCA projection from 3D to 2D.

4.1.1 Example: Reduce 2D data to 1D

First of all, let's center the points around the mean along x_1 and x_2.

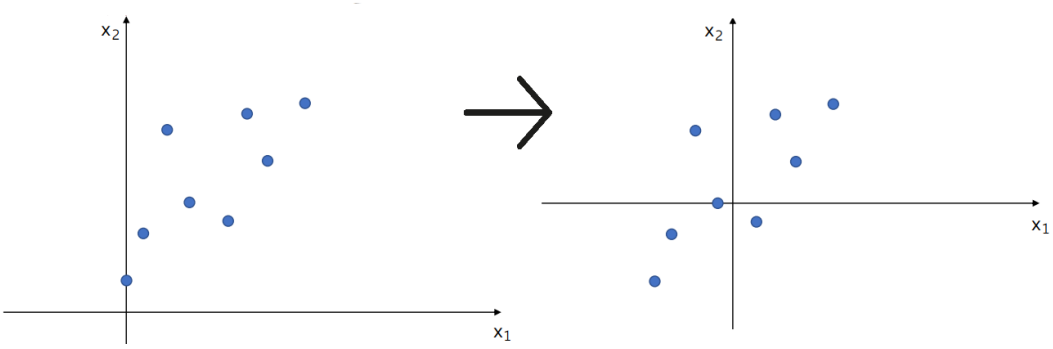


Figure 18: First step of PCA.

We can project x_1 and x_2 onto infinitely many one-dimensional axes e. However, we now focus on two specific directions, e_1 and e_2, as illustrated in figure 19. Points projected onto e_1 look more spread-out than onto e_2. In other words, The variance along e_1 is larger than along e_2.

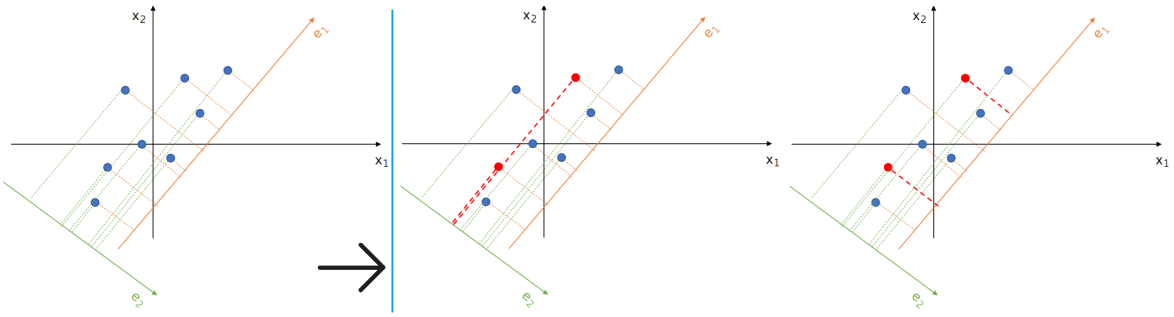


Figure 19: First step of PCA.

Now consider the 2 red points. They actually look indistinguishable along e_2, while they better preserve their distance along e_1.

We aim to choose the projection axis e in such a way that it preserves the original structure of the data, minimizing the risk that distant points in the original space appear close in the lower-dimensional space. This is achieved by maximizing the variance of the data projected onto e.

4.1.2 Variance and Covariance of Random Variables

The **Variance** of a random variable X measures how far a set of (random) values are spread out from their mean. Formally, it is defined as the expected value of the squared deviation from the mean:

$$\mathrm{Var}(X) = \mathbb{E}[(X - \mu)^2]$$

where $\mu = \mathbb{E}[X]$.

Covariance measures the joint variability of two random variables X and Y. Specifically, it indicates whether X and Y tend to increase or decrease together, or if one increases while the other decreases. Formally, the covariance is defined as the expected value of the product of their deviations from their respective means:

$$\mathrm{Cov}(X, Y) = \mathbb{E}\big[(X - \mu_X)(Y - \mu_Y)\big]$$

note that $\mathrm{Cov}(X, X) = \mathrm{Var}(X)$.

Given a random vector $\mathbf{X} = (X_1, \ldots, X_d)$, its **Covariance Matrix K** is a $d \times d$ square matrix where each entry represents the covariance between pairs of elements:

$$K[i, j] = \mathrm{Cov}(X_i, X_j)$$

the diagonal entries correspond to the variances of each element, since they result from the covariance of each variable with itself.

For example, when $d = 2$, $\mathbf{X} = (X_1, X_2)$ and the covariance matrix $\mathbf{K}$ is a 2×2 matrix K. To simplify covariance calculations, each data point can be centered by subtracting the mean of each dimension so that the mean of every dimension is zero. Let n be the total number of data points $x_1, \ldots, x_n$, where each data point x_i is represented by $(x_{i,1}, x_{i,2})$ pairs. We associate 2 random variables X_1, X_2 to each dimension, and we compute:

$$\mu_1 = \mathbb{E}[X_1] = \frac{1}{n} \sum_{i=1}^{n} x_{i,1}$$

$$\mu_2 = \mathbb{E}[X_2] = \frac{1}{n} \sum_{i=1}^{n} x_{i,2}$$

$$x_i = (x_{i,1} - \mu_1, x_{i,2} - \mu_2)$$

Let's now rewrite each data point x_i as follows:

$$x_i = (x'_{i,1}, x'_{i,2})$$

$$\text{where} \quad x'_{i,1} = x_{i,1} - \mu_1 \quad \text{and} \quad x'_{i,2} = x_{i,2} - \mu_2$$

$$\mu_1^{\text{new}} = \mathbb{E}[X_1] = \frac{1}{n} \sum_{i=1}^{n} x'_{i,1}$$

$$\mu_2^{\text{new}} = \mathbb{E}[X_2] = \frac{1}{n} \sum_{i=1}^{n} x'_{i,2}$$

And then we scale data so as to have 0-mean on all dimensions:

$$\mu_1^{\text{new}} = 0 \quad \text{and} \quad \mu_2^{\text{new}} = 0$$

This allows to compute covariance much easily, and as a consequence the covariance matrix gets easier too:

$$\mathrm{Cov}(X_1, X_2) = \mathbb{E}\big[(X_1 - \mu_1^{\text{new}})(X_2 - \mu_2^{\text{new}})\big] = \mathbb{E}\big[X_1 X_2\big] \quad \text{since} \quad \mu_1^{\text{new}} = \mu_2^{\text{new}} = 0$$

Starting from a covariance matrix, we can begin with a random vector $\mathbf{e}$ and multiply it by the matrix to obtain a new vector $\mathbf{e}_1$. By repeating this process iteratively with the new vector, the resulting vectors grow longer and increasingly align with the direction of the largest variance, as illustrated in the figure. In linear algebra the converged vector is also a eigenvector.

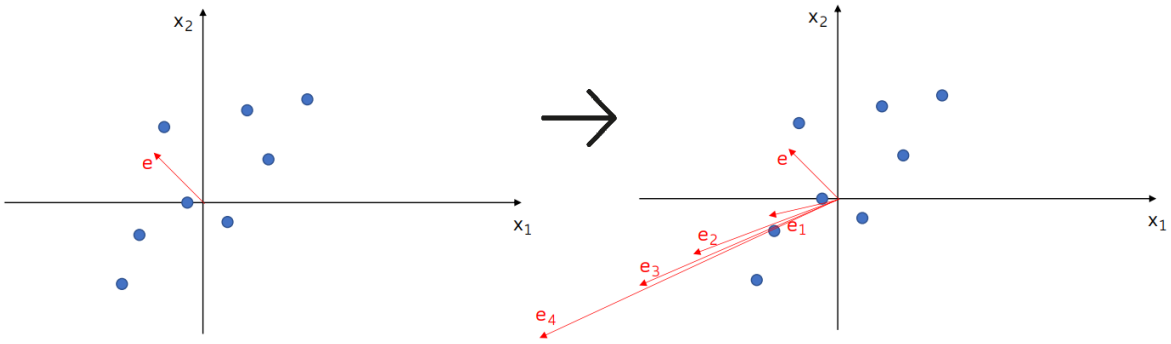


Figure 20: Convergence of eigenvectors.

4.1.3 Eigenvectors as Principal Components

Given a square matrix K, a non-zero vector $\mathbf{e}$ is called an **eigenvector** of K if there exists a scalar λ such that:

$$K\mathbf{e} = \lambda\mathbf{e}.$$

The scalar λ is called the **eigenvalue** corresponding to the eigenvector $\mathbf{e}$.

When a covariance matrix K multiplies one of its eigenvectors $\mathbf{e}$, the direction of $\mathbf{e}$ remains unchanged and the vector is scaled by a factor λ, the corresponding eigenvalue.

To compute eigenvectors, we solve the equation $K\mathbf{e} = \lambda\mathbf{e}$ rewriting it as:

$$(K - \lambda I)\mathbf{e} = 0$$

where I is the identity matrix. To solve this homogeneous system, since homogeneous systems always admit the trivial solution $\mathbf{e} = 0$, we have to find a non-trivial solution in which $(K - \lambda I)$ is non-invertible, otherwise:

$$(K - \lambda I)(K - \lambda I)^{-1}\mathbf{e} = 0(K - \lambda I)^{-1} \implies \mathbf{e} = 0$$

Since a square matrix is invertible if and only if its determinant is non-zero, the matrix $(K - \lambda I)$ is not invertible when $\det(K - \lambda I) = 0$. Only in this case, the corresponding homogeneous system admits a non-trivial solution.

For instance, if we want to compute the eigenvectors of a covariance matrix:

1. find the eigenvalues by solving $det(K - \lambda I) = 0$:

$$det(\begin{bmatrix} 2-\lambda & 4/5 \\ 4/5 & 3/5-\lambda \end{bmatrix}) = 0 \implies (2-\lambda)(3/5-\lambda) - (4/5)(4/5) = 0$$

$$\implies \lambda^2 - 13/5\lambda + 14/25 = 0$$

$$\implies \lambda_1 = \frac{13+\sqrt{113}}{10} \approx 2.36 \quad \text{and} \quad \lambda_2 = \frac{13-\sqrt{113}}{10} \approx 0.24$$

2. plug each eigenvalue in to find the corresponding eigenvector:

$$K \quad \mathbf{e} \quad = \lambda \quad \mathbf{e}$$

$$\begin{bmatrix} 2 & 4/5 \\ 4/5 & 3/5 \end{bmatrix} \begin{bmatrix} e_{1.1} \\ e_{1,2} \end{bmatrix} = \lambda_1 \begin{bmatrix} e_{1.1} \\ e_{1,2} \end{bmatrix}$$

$$\begin{bmatrix} 2 & 4/5 \\ 4/5 & 3/5 \end{bmatrix} \begin{bmatrix} e_{2.1} \\ e_{2,2} \end{bmatrix} = \lambda_2 \begin{bmatrix} e_{2.1} \\ e_{2,2} \end{bmatrix}$$

3. see what happens for λ_1:

$$\begin{cases} 2e_{1,1} + 4/5e_{1,2} = \frac{13+\sqrt{113}}{10} e_{1,1} \\ 4/5e_{1,1} + 3/5e_{1,2} = \frac{13+\sqrt{113}}{10} e_{1,2} \end{cases} \quad \begin{cases} 2e_{1,1} + 0.8e_{1,2} = 2.36e_{1,1} \\ 0.8e_{1,1} + 0.6e_{1,2} = 2.36e_{1,2} \end{cases} \quad \Rightarrow e_{1.1} \approx 2.2e_{1,2}$$

Any vector which satisfies the relationship $e_{1.1} \approx 2.2e_{1,2}$ is an eigenvector, but by convention we prefer to restrict $\|\mathbf{e}_1\| = \sqrt{e_{1,1}^2 + e_{1,2}^2} = 1$ (normalization).

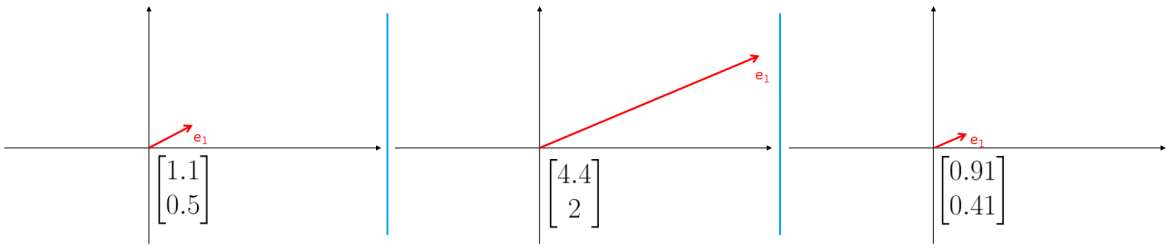


Figure 21: Different eigenvectors from the same eigenvalue.

4. The second eigenvector $\mathbf{e}_2$ can be found by plugging in the smaller eigenvalue λ_2, and it will be just orthogonal to the previously found $\mathbf{e}_1$:

5. The two eigenvectors become the new coordinate system replacing the original ones. Obviously $\mathbf{e}_1$ is the first principal component and $\mathbf{e}_2$ is the second principal component.

For any $d \times d$ real symmetric matrix K, all eigenvalues are real and the eigenvectors can be chosen to be real and orthonormal.

In order to represent a point x from the original space in the new ($\mathbf{e}_1$, $\mathbf{e}_2$)-coordinate system:

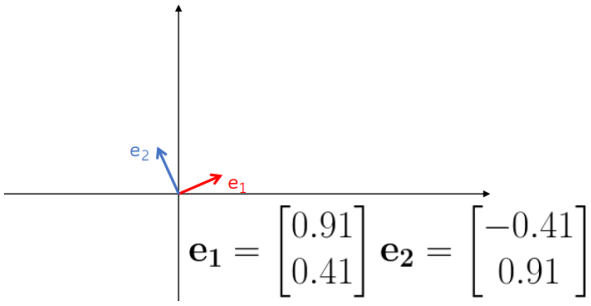


Figure 22: $\mathbf{e}_1$ and $\mathbf{e}_2$ once computed.

1. Center x around the mean of each dimension:

$$x' = x - \mu = (x_1 - \mu_1, x_2 - \mu_2)$$

2. Project x' on each new dimension:

$$x' = (x'_1, x'_2) = (x'^T \mathbf{e}_1, x'^T \mathbf{e}_2)$$

where (x'_1, x'_2) are the coordinates of x' in the $(\mathbf{e}_1, \mathbf{e}_2)$-space.

The dot product is used because it captures the projection of x' onto $\mathbf{e}_1$ and $\mathbf{e}_2$:

$$\cos\theta = \frac{x' \cdot \mathbf{e}_1}{\|x'\|\|\mathbf{e}_1\|} \quad \Rightarrow \quad \|x'\|\cos\theta = \frac{x' \cdot \mathbf{e}_1}{\|\mathbf{e}_1\|} \quad \Rightarrow \quad \|x'\|\cos\theta = x' \cdot \mathbf{e}_1$$

since $\|\mathbf{e}_1\| = 1$ and the same reasoning applies to $\mathbf{e}_2$. At the end of the process,

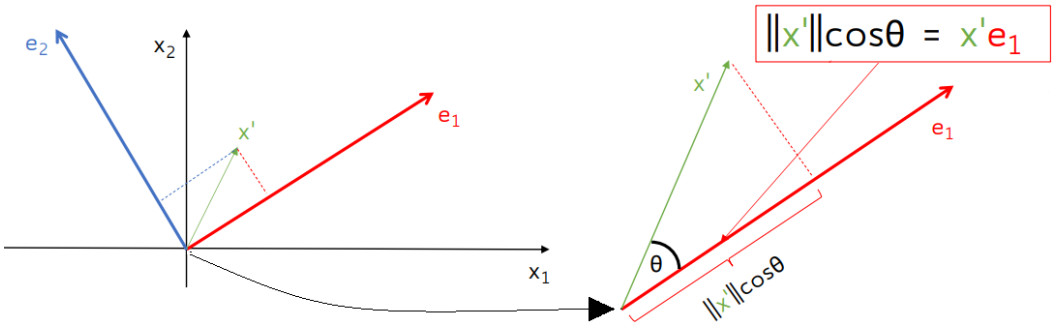


Figure 23: Projecting x' onto $\mathbf{e}_1$.

we finally have the new coordinates of the original data point x according to the eigenvectors $\mathbf{e}_1$ and $\mathbf{e}_2$:

$$x' = \begin{bmatrix} x'_1 \\ x'_2 \end{bmatrix} = \begin{bmatrix} x'^T \mathbf{e}_1 \\ x'^T \mathbf{e}_2 \end{bmatrix} = \begin{bmatrix} (x_1 - \mu_1)e_{1,1} + (x_2 - \mu_2)e_{1,2} \\ (x_1 - \mu_1)e_{2,1} + (x_2 - \mu_2)e_{2,2} \end{bmatrix}$$

In the more general case of d dimensions, let $x = (x_1, \ldots, x_d)^T$ and let $\mathbf{e}_1, \ldots, \mathbf{e}_k \in \mathbb{R}^d$ be the first k principal components, where $k \ll d$. The mean-centering step

becomes:

$$\mathbf{x}' = \mathbf{x} - \boldsymbol{\mu} = \begin{bmatrix} x_1 - \mu_1 \\ x_2 - \mu_2 \\ \vdots \\ x_d - \mu_d \end{bmatrix}$$

while the projection of $\mathbf{x}'$ onto the principal components is given by:

$$x' = \begin{bmatrix} x'_1 \\ \vdots \\ x'_k \end{bmatrix} = \begin{bmatrix} (x-\mu)^T \mathbf{e}_1 \\ \vdots \\ (x-\mu)^T \mathbf{e}_k \end{bmatrix} = \begin{bmatrix} (x_1 - \mu_1)e_{1,1} + \cdots + (x_d - \mu_d)e_{1,d} \\ \vdots \\ (x_1 - \mu_1)e_{k,1} + \cdots + (x_d - \mu_d)e_{k,d} \end{bmatrix}$$

We choose the eigenvector associated with the largest eigenvalue because it maximizes the variance of the data when projected onto that direction. In fact, the eigenvalue λ represents the variance along its corresponding eigenvector, and the one with the largest λ captures the direction of greatest spread in the data.

In a d-dimensional space, PCA provides d orthonormal eigenvectors $\mathbf{e}_1, \ldots, \mathbf{e}_d$, each associated with an eigenvalue λ_i representing the variance along that direction. To reduce dimensionality, we select the top $k \ll d$ eigenvectors corresponding to the largest eigenvalues, since they collectively capture the majority of the total variance.

To determine the number of dimensions to retain, we first sort the eigenvectors by decreasing eigenvalues $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_d$. Then, we select the top k eigenvectors such that the cumulative variance $\sum_{i=1}^{k} \lambda_i$ accounts for a desired percentage (e.g., $90-95\%$) of the total variance $\sum_{i=1}^{d} \lambda_i$.

4.1.4 Issues with PCA

Since variance and covariance are sensitive to scale, dimensions with disproportionately large values can dominate and be wrongly identified as principal components. To prevent this, each feature is standardized to have zero mean and unit variance using $z = \frac{x-\mu}{\sigma}$ where σ represents the standard deviation of x.

PCA also assumes that the data lie near a linear subspace, meaning the projection is onto a flat hyperplane (e.g., a line in 1D, a plane in 2D). However, if the data instead lie on a nonlinear low-dimensional manifold, PCA may not work well.

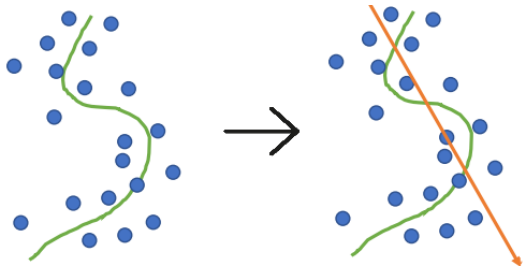


Figure 24: PCA vs non-linearity.

5 Recommender Systems

With the explosion of content and services on the Web, users are overwhelmed by too many choices. To face this phenomenon known as **Information Overload**, we try to make this vast amount of information manageable through techniques like searching, filtering, and recommendation systems.

We are constantly moving from scarcity to abundance of information. **Recommender Systems** are essential for filtering this abundance and matching users with the most relevant products or services, ultimately boosting satisfaction and loyalty.

In an economy of abundance, a "long tail" represents a vast number of niche or less popular items that collectively hold significant value. A good recommendation system should help uncover and promote these hidden items, expanding user choice beyond the most obvious hits.

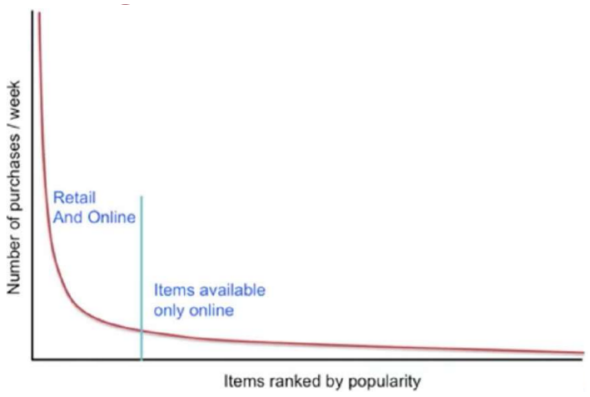


Figure 25: The long tail in the economics of abundance.

Let's define a formalism of recommender systems. Let $U = \{u_1, \ldots, u_m\}$ be the set of users, and $I = \{i_1, \ldots, i_n\}$ be the set of items. We define a utility function:

$$r : U \times I \to \mathcal{R}$$

where $\mathcal{R} \subseteq \mathbb{R}$ is the set of possible ratings totally ordered, and $r(u, i)$ represents the degree of preference or satisfaction that user $u \in U$ associates with item $i \in I$.

		MOVIES							
									
USERS	Alice	2		5	4	5	4		4
	Bob	4					3		3
	Carl	5	5	3	4	5	4		5
	...	...	...	...	...	...	...	...	...
	Zoe		1	3				5	4

Figure 26: The utility function as a user-item matrix.

The set $\mathcal{R}$ can take the form of a discrete set of values ($\mathcal{R} = \{0, 1, \ldots, v-1\}$) or the form of a continuous interval ($\mathcal{R} = [0, 1]$).

The utility function r can also be represented as a user-item matrix as shown in figure 26.

A recommender system faces three key challenges:

- **data collection**, which involves gathering known ratings to populate the utility matrix,
- **rating prediction**, which aims to infer unknown ratings based on known ones,
- **recommendation evaluation**, which assesses the effectiveness of the recommendation methods.

Data collection can be either **explicit** or **implicit**. The explicit method involves asking users to rate items directly, while the implicit one consists of learning from user behavior and interactions. Unfortunately, explicit feedback is not very scalable, as only a small fraction of users typically provide ratings.

In rating prediction, the utility matrix is sparse, as most users haven't rated most items. Moreover, there's the cold start problem, where new users or items lack historical data.

Recommendation evaluation includes various aspects: **RMSE** (Root Mean Squared Error) assesses how close predicted ratings are to actual ones, **personalization** checks whether recommendations are tailored to individual users, **serendipity** captures the ability to suggest unexpected but relevant items, and **MAP@K/MAR@K** evaluate how well the system ranks relevant items among the top suggestions.

There are three main approaches to recommender systems:

- **Content-based filtering**: recommends items similar to those the user liked in the past.
- **Collaborative filtering**: leverages user-item interactions to make recommendations. It can be divided into:
  - Neighborhood-based (also known as memory-based): based on similarities between users or items.
  - Latent factor-based (also known as model-based): uses matrix factorization or machine learning models to uncover hidden patterns.
  - Hybrid (memory-model-based) combines memory-based and model-based approaches for improved accuracy and robustness.
- **Hybrid methods**: combine content-based and collaborative filtering strategies, to enhance performance.

5.1 Content-Based Filtering

Content-based filtering recommends items to a user u by identifying those that are similar to items the user has rated highly in the past. This approach relies on the concept of item/user profiles.

The process typically involves the following steps:

1. Build item profiles (i.e., a description of items using metadata),
2. Based on the item profiles, build user profiles (what the user likes),
3. Match the user profile with the item catalog.

Building item profiles means representing each item as a set of descriptive features; such as genre, director, or title for movies, or age, job, and social connections for people. Each profile can be thought of as a vector of numerical or categorical attributes used for comparison and recommendation.

The goal of user profiling is to create a profile for each user u based on the features of the items they have rated. Let $I_u \subseteq I$ be the set of items rated by user u, and let $\mathbf{i}_j$ be the vector profile of item j. The simplest approach to constructing a user profile is to compute the average of the profiles of the items the user has rated:

$$\mathbf{u}_i = \frac{1}{|I_u|} \sum_{\mathbf{i}_j \in I_u} \mathbf{i}_j$$

this resulting vector $\mathbf{u}_i$ summarizes the user's preferences based on past interactions, and treats all items equally, independently of the rating.

Once a user profile $\mathbf{u}$ has been built, it can be used to estimate the missing entries in the utility matrix. To do this, for each unrated item, we compute the similarity between the user's profile vector and the item's profile vector. This is typically done using measures such as cosine similarity or Pearson's correlation. After computing similarity scores for all unrated items, the system selects the top-k items with the highest similarity values and recommends them to the user as the most relevant suggestions.

For the top-k recommendation, we define an iterative process as follows:

- Let $A^0 = \emptyset$ be the initial empty set.
- Then we select the most similar item (not yet rated or recommended):

$$A^1 = \arg\max_i \left\{ \text{sim}(u, i) \mid i \in I - I_u - A^0 \right\}$$

- At each step j, we pick the next most similar item not already chosen or rated:

$$A^j = \arg\max_i \left\{ \text{sim}(u, i) \mid i \in I - I_u - \bigcup_{l=0}^{j-1} A^l \right\}$$

- After k iterations, the set of top-k recommended items for user u is:

$$R_{u,k} = \bigcup_{j=1}^{k} A^j = \bigcup_{j=1}^{k} \arg\max_i \left\{ \text{sim}(u,i) \mid i \in I - I_u - \bigcup_{l=0}^{j-1} A^l \right\}$$

Content-based filtering offers several advantages: it relies solely on a user's own ratings, making it independent from other users' data, and it avoids the item cold start problem, since even new or unrated items can be recommended based on content similarity. Moreover, recommendations are explainable, as they are grounded in identifiable item features.

At the same time, it also presents some limitations: it depends heavily on the quality and availability of item features, and it tends to suffer from overspecialization, recommending only items similar to those already known. It also cannot leverage other users' opinions and struggles with the cold start problem for new users who haven't rated any item, often requiring the use of average profiles that are updated over time.

5.2 Collaborative Filtering

Collaborative filtering recommends items by leveraging the preferences of other users with similar tastes, based on user-to-user or item-to-item similarity. Unlike content-based methods, it does not require explicit profiles.

5.2.1 Neighborhood-Based

As already discussed, there are three approaches to collaborative filtering; the first is the **neighborhood-based** approach. It computes similarities between users or items to make predictions. In the **user-based** approach, a user's preference for an item is estimated from the ratings given by similar users, while in the **item-based** approach, the user's preference is inferred from the user's ratings of similar items.

User-Based Collaborative Filtering: Given a user u and an item i that has not been rated by u, the goal is to estimate the rating $r(u,i)$. To do this, we first consider the set of users $\{u' \mid u' \neq u\}$ who have already rated item i. From this set, we extract a subset of k users, known as the k-neighborhood of u, based on the similarity of their rating patterns with u. This similarity is computed directly from user ratings, without the need for explicit user profiles. Once the neighborhood is identified, the estimated rating $r(u,i)$ is computed by aggregating the ratings for item i provided by the k most similar users to u.

In theory, the prediction of the rating $r(u,i)$ could be computed for any item i that has not yet been rated by user u. However, in practice, we are mainly interested in estimating $r(u,i)$ only for those items that have already been rated by users in the k-neighborhood of u. The intuition behind this is that if a user v does not belong to

the k-neighborhood of u, it is unlikely that u will be interested in items that only v has rated. In other words, the k-neighborhood of u must first be identified in order to restrict the set of items for which we attempt to predict ratings. This helps focus the recommendation process on the most relevant subset of items.

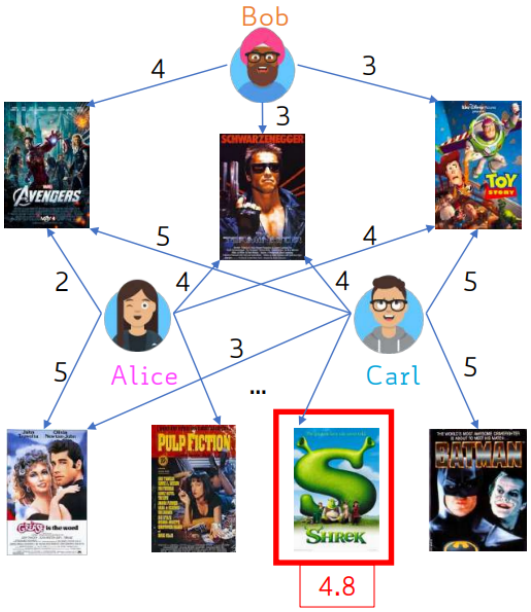


Figure 27: User-based neighborhood example.

The user-to-user similarity is computed directly from user ratings. Intuitively, two users u_1 and u_2 are considered similar if they have given similar ratings to the same items. Each user can be represented as a vector $\mathbf{r}_u$ in the item-rating space, and the similarity between users is then measured by comparing these rating vectors using appropriate similarity metrics.

One possible metric to compute user-to-user similarity is the **Jaccard similarity**, defined as:

$$\text{sim}(u, v) = J(\mathbf{r}_u, \mathbf{r}_v) = \frac{|\mathbf{r}_u \cap \mathbf{r}_v|}{|\mathbf{r}_u \cup \mathbf{r}_v|}$$

unfortunately, as illustrated in figure 28, it completely ignores the rating values, which generally is not what we want.

$$\text{sim}(\text{Alice}, \text{Carl}) = \frac{|\mathbf{r}_{\text{Alice}} \cap \mathbf{r}_{\text{Carl}}|}{|\mathbf{r}_{\text{Alice}} \cup \mathbf{r}_{\text{Carl}}|}$$

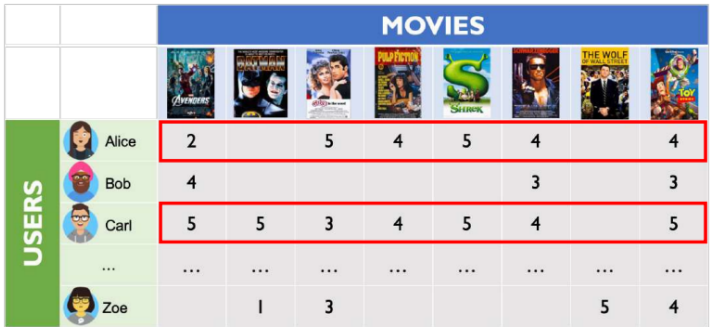
$$= \frac{6}{7} \approx 0.86$$

Figure 28: Jaccard similarity example.

A further considerable metric is the **cosine similarity**, where:

$$\text{sim}(u, v) = \text{cosine}(\mathbf{r}_u, \mathbf{r}_v) = \frac{\mathbf{r}_u \cdot \mathbf{r}_v}{||\mathbf{r}_u|| ||\mathbf{r}_v||}$$

even this metric is not perfect, since missing rating values are treated as 0s, resulting in negative effects.



$$\text{sim}(\text{Alice}, \text{Carl}) = \frac{\mathbf{r}_{\text{Alice}} \cdot \mathbf{r}_{\text{Carl}}}{||\mathbf{r}_{\text{Alice}}|| ||\mathbf{r}_{\text{Carl}}||}$$

$$= \frac{102}{\sqrt{102}\sqrt{141}} \approx 0.85$$

Figure 29: Cosine similarity example.

The solution to the problem of cosine similarity is provided by **Pearson similarity**, which normalizes the ratings by subtracting the user's average rating from each value:

$$\text{sim}(u, v) = \text{Pearson}(\mathbf{r}_u, \mathbf{r}_v) = \frac{(\mathbf{r}_u - \bar{\mathbf{r}}_u) \cdot (\mathbf{r}_v - \bar{\mathbf{r}}_v)}{\sqrt{(\mathbf{r}_u - \bar{\mathbf{r}}_u)^T \cdot (\mathbf{r}_u - \bar{\mathbf{r}}_u)} \sqrt{(\mathbf{r}_v - \bar{\mathbf{r}}_v)^T \cdot (\mathbf{r}_v - \bar{\mathbf{r}}_v)}}$$

in this way 0s become neutral.

In general, let $\mathcal{U}^k$ be the set of top-k most "similar" users to u, and $\mathcal{I}^k$ the set of items rated by u's neighbors:

$$\mathcal{I}^k = \left\{ i \in \mathcal{I} \ : \ \exists \, u' \in \mathcal{U}^k \text{ s.t. } \mathbf{r}_{u',i} \text{ is known} \right\}$$

where $\mathbf{r}_{u,i} = \mathbf{r}_u[i] = \mathbf{r}(u, i)$ is the predicted rating given by user u to item i.

There are two common strategies to compute the predicted rating $\mathbf{r}_{u,i}$:

- Simple average of neighbor ratings:

$$\mathbf{r}_{u,i} = \frac{1}{k} \sum_{u' \in \mathcal{U}^k} \mathbf{r}_{u',i}$$

- Similarity-weighted average:

$$\mathbf{r}_{u,i} = \frac{1}{k} \sum_{u' \in \mathcal{U}^k} \text{sim}(u, u') \cdot \mathbf{r}_{u',i}$$

The second approach takes into account how similar each neighbor u' is to the target user u, giving more weight to the opinions of more similar users.

User-based collaborative filtering faces several challenges:

- **Sparsity** arises when the system contains a large number of items but very few ratings, making it difficult to find sufficient overlap between users.
- **Efficiency** is another concern, as computing similarities between all pairs of users can be computationally expensive, especially as the user base grows.
- Finally, **aging** is a problem because user preferences can change over time, requiring frequent recomputation of the entire similarity model to stay up to date.

Item-Based Collaborative Filtering: Item-based collaborative filtering was introduced by Amazon as a solution to the main limitations of user-based approaches. Since recommendation systems usually have far more users than items, each item tends to receive more ratings, making its average rating more stable over time. As a result, the item-based model is less affected by aging and does not require frequent recomputation.

Given a user u and an item i not yet rated by u, we want to estimate the rating $r(u, i)$. To do so, we consider the set I_u of items already rated by u, and identify a subset of k items most similar to i; its k-neighborhood. The similarity between items is computed without relying on explicit content or metadata, but rather based on user ratings. The predicted rating $r(u, i)$ is then estimated by aggregating u's ratings on these similar items.

In this case, the assumption behind similarity is that two items i_1 and i_2 are similar if they are rated similarly by similar users. Each item is represented as a vector of user ratings, and the similarity between items is computed in this user-rating space.

For example, suppose we want to predict the rating Bob would give to Shrek. We first extract the subset of k most similar items to Shrek which have been rated by Bob. In figure 30 we assume those are "The Avengers" and "The Terminator". The

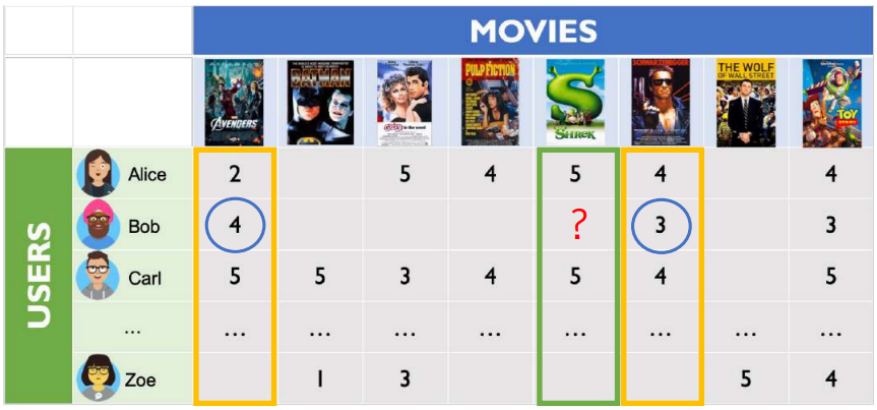


Figure 30: Item-based neighborhood example.

predicted rating is computed as an aggregating function of the ratings that Bob gave to the k most similar movies to Shrek.

Let $\mathbf{r}_i$ be the vector of ratings given to i, let $\mathcal{I}_u$ be the set of items rated by u, and:

$$\mathcal{I}_u^k = \text{argmax}_{\mathcal{I}'_u \subseteq \mathcal{I}_u, |\mathcal{I}'_u|=k} \sum_{i' \in \mathcal{I}'_u} \text{sim}(i, i')$$

$\mathcal{I}_u^k$ is the set of top-k most "similar" items to i among those rated by u.

Common similarity measures include the Jaccard index and cosine similarity, often normalized to correspond to Pearson correlation. Once similarities are established, rating predictions can be made using the same approaches as in user-based collaborative filtering: either by taking a simple average of the ratings on similar items, or by computing a weighted average where each rating is weighted by the similarity between items.$\mathbf{r}_{u,i}$:

- Simple average of neighbor ratings:

$$\mathbf{r}_{u,i} = \frac{1}{k} \sum_{i' \in \mathcal{I}_u^k} \mathbf{r}_{u,i'}$$

- Similarity-weighted average:

$$\mathbf{r}_{u,i} = \frac{1}{k} \sum_{i' \in \mathcal{I}_u^k} \text{sim}(i, i') \cdot \mathbf{r}_{u,i'}$$

In general, item-based works better than user-based CF. The most computationally expensive step is identifying the top-k most similar users or items. Performing this similarity search online (i.e., in real time for every user or item) is generally too costly. Therefore, the computation of nearest neighbors is typically done offline in advance. This process involves solving the k-nearest neighbors problem in high-dimensional spaces.

But in high-dimensional spaces, traditional nearest-neighbor search becomes inefficient due to the **curse of dimensionality**, where distance metrics lose discriminative power. To address this, **Locality-Sensitive Hashing** (LSH) is used as an efficient approximation technique that hashes similar items into the same buckets, enabling faster retrieval of similar users or items in memory-based collaborative filtering.

5.2.2 Latent Factor-Based

Latent factor models try to predict ratings by representing both items and users with a number of hidden factors inferred from observed ratings.

Matrix factorization is a popular and effective approach in latent factor models; the idea is that both users and items are represented as vectors in a lower-dimensional latent space. These vectors are learned from observed item ratings, allowing the model to capture hidden patterns and preferences.

Matrix Factorization Framework: Each user u is associated with a vector $\mathbf{x}_u \in \mathbb{R}^d$, and each item i with a vector $\mathbf{w}_i \in \mathbb{R}^d$. Moreover, $x_u[k]$ represents the degree to which user u is interested in the k-th latent factor, while $w_i[k]$ indicates how strongly item i expresses that same factor.

The latent factors represent hidden features that describe both users and items; for example, in a movie setting, one factor might capture a user's affinity for Disney movies or how much a movie resembles that style. These features are not predefined or explicitly labeled, which is why they are called latent features.

Given a user u and an item i, the actual rating is denoted as $r_{u,i}$. The predicted rating, instead, is computed using the inner product of their latent factor vectors:

$$\hat{r}_{u,i} = \mathbf{x}_u^T \cdot \mathbf{w}_i = \sum_{j=1}^{d} x_{u,j} \cdot w_{j,i}$$

where $\mathbf{x}_u \in \mathbb{R}^d$ is the latent vector representing user u and $\mathbf{w}_i \in \mathbb{R}^d$ is the latent vector representing item i.

The main challenge lies in learning the appropriate mapping of users and items into their respective latent factor representations, $\mathbf{x}_u$ and $\mathbf{w}_i$, from the observed data. Once these vectors are learned, recommendations for a given user are generated by computing the predicted ratings $\hat{r}_{u,i}$ for all items not yet rated by the user, and selecting the top-k items with the highest predicted scores.

To learn the latent factor representations $\mathbf{x}_u$ and $\mathbf{w}_i$, we assume access to a dataset of observed ratings. This corresponds to a partially filled user-item rating matrix R, where each entry $r_{u,i}$ is known only for some pairs $(u, i) \in \mathcal{D}$, the training set.

The learning process consists in minimizing the following regularized squared error loss function:

$$L(X, W) = \sum_{(u,i)\in\mathcal{D}} \left(r_{u,i} - \mathbf{x}_u^T \cdot \mathbf{w}_i\right)^2 + \lambda \left(\sum_{u\in\mathcal{D}} \|\mathbf{x}_u\|^2 + \sum_{i\in\mathcal{D}} \|\mathbf{w}_i\|^2\right)$$

where the first term represents the squared error between the actual and predicted ratings, while the second is a regularization term that prevents overfitting by penalizing large values in the latent factor vectors. The hyperparameter λ controls the trade-off between accuracy on the training data and model complexity.

Thus, we want to find:

$$X^*,\ W^* = \arg\min_{X,W} L(X, W) =$$

$$\arg\min_{X,W} \sum_{(u,i)\in\mathcal{D}} \left(r_{u,i} - \mathbf{x}_u^T \cdot \mathbf{w}_i\right)^2 + \lambda \left(\sum_{u\in\mathcal{D}} \|\mathbf{x}_u\|^2 + \sum_{i\in\mathcal{D}} \|\mathbf{w}_i\|^2\right) =$$

$$\arg\min_{X,W} \frac{1}{2} \sum_{(u,i)\in\mathcal{D}} \left(r_{u,i} - \mathbf{x}_u^T \cdot \mathbf{w}_i\right)^2 + \lambda \left(\sum_{u\in\mathcal{D}} \|\mathbf{x}_u\|^2 + \sum_{i\in\mathcal{D}} \|\mathbf{w}_i\|^2\right)$$

where $\frac{1}{2}$ is often introduced to simplify the derivative of the quadratic term.

To learn the latent factor matrices X and W, two main optimization methods are commonly used:

- **Stochastic Gradient Descent (SGD)**:
  For each training instance (u, i), the gradient of the loss with respect to $\mathbf{x}_u$ and $\mathbf{w}_i$ is computed.

$$\nabla L(\mathbf{x}_u; \mathbf{w}_i) = \frac{1}{2}\Big[-2(r_{u,i} - \mathbf{x}_u^T \cdot \mathbf{w}_i) \cdot \mathbf{w}_i + 2\lambda \mathbf{x}_u \Big] = -(r_{u,i} - \mathbf{x}_u^T \cdot \mathbf{w}_i) \cdot \mathbf{w}_i + \lambda \mathbf{x}_u$$
$$\nabla L(\mathbf{w}_i; \mathbf{x}_u) = \frac{1}{2}\Big[-2(r_{u,i} - \mathbf{x}_u^T \cdot \mathbf{w}_i) \cdot \mathbf{x}_u + 2\lambda \mathbf{w}_i \Big] = -(r_{u,i} - \mathbf{x}_u^T \cdot \mathbf{w}_i) \cdot \mathbf{x}_u + \lambda \mathbf{w}_i$$

  The updating strategy adopted by SGD is as follows:

$$\mathbf{x}_u^{(t+1)} \leftarrow \mathbf{x}_u^{(t)} - \eta \nabla L(\mathbf{x}_u^{(t)}; \mathbf{w}_i^{(t)})$$
$$\mathbf{w}_i^{(t+1)} \leftarrow \mathbf{w}_i^{(t)} - \eta \nabla L(\mathbf{w}_i^{(t)}; \mathbf{x}_u^{(t)})$$

  where $\mathbf{x}_u^{(0)}$ and $\mathbf{w}_i^{(0)}$ are typically randomly initialized.

  At each iteration, both user and item latent vectors are updated by a magnitude proportional to η in the opposite direction of the gradient, since this is a minimization problem.

  We also define the prediction error associated with each training instance (u, i):

$$e_{u,i} = r_{u,i} - \mathbf{x}_u^T \cdot \mathbf{w}_i \quad \text{thus}$$
$$\nabla L(\mathbf{x}_u; \mathbf{w}_i) = -e_{u,i} \cdot \mathbf{w}_i + \lambda \mathbf{x}_u \quad \text{and}$$
$$\mathbf{x}_u^{(t+1)} \leftarrow \mathbf{x}_u^{(t)} + \eta(e_{u,i} \cdot \mathbf{w}_i^{(t)} - \lambda \mathbf{x}_u^{(t)})$$

  and the reasoning is analogous for $\nabla L(\mathbf{w}_i; \mathbf{x}_u)$ and $\mathbf{w}_i^{(t+1)}$.

  SGD is effective for optimizing matrix factorization models, but becomes less practical when the rating matrix R is high-dimensional. This is due to the need to optimize $d(m+n)$ parameters, which in real-world scenarios can become very large, often necessitating parallelization or alternative optimization strategies.

- **Alternating Least Squares (ALS)**
  Starting from the assumption that the loss function is non-convex, since both $\mathbf{x}_u$ and $\mathbf{w}_i$ are unknown and thus variable, the idea behind ALS is to fix one of the two latent vectors and update the other. When one of the two is fixed, the problem becomes quadratic (and therefore convex), making it optimizable. In this case, the optimization reduces to a traditional least squares problem, which can be solved using OLS or its regularized variant (e.g., pseudo-inverse). The roles of the two vectors are alternated iteratively, allowing both to be progressively updated.

Let's assume we fix the item latent vector $\mathbf{w}_i$ and we take the gradient with respect to the user latent vector $\mathbf{x}_u$:

$$\nabla L(\mathbf{x}_u; \mathbf{w}_i) = \frac{1}{2}\Big[-2 \sum_{i \in \mathcal{D}} (r_{u,i} - \mathbf{x}_u^T \cdot \mathbf{w}_i) \cdot \mathbf{w}_i + 2\lambda \mathbf{x}_u \Big] =$$

$$- \sum_{i \in \mathcal{D}} (r_{u,i} - \mathbf{x}_u^T \cdot \mathbf{w}_i) \cdot \mathbf{w}_i + \lambda \mathbf{x}_u \quad \text{and we want to set}$$

$$- \sum_{i \in \mathcal{D}} (r_{u,i} - \mathbf{x}_u^T \cdot \mathbf{w}_i) \cdot \mathbf{w}_i + \lambda \mathbf{x}_u = 0$$

$$= -W^T (\mathbf{r}_u - W \cdot \mathbf{x}_u) + \lambda \mathbf{x}_u = 0$$

$$= W^T \cdot \mathbf{r}_u = W^T W \cdot \mathbf{x}_u + \lambda \mathbf{x}_u$$

$$= W^T \cdot \mathbf{r}_u = \mathbf{x}_u \Big(W^T W + \lambda I \Big)$$

$$= \Big(W^T W + \lambda I \Big)^{-1} \cdot W^T \cdot \mathbf{r}_u = \mathbf{x}_u \Big(W^T W + \lambda I \Big) \cdot \Big(W^T W + \lambda I \Big)^{-1}$$

$$\mathbf{x}_u = \Big(W^T W + \lambda I \Big)^{-1} \cdot W^T \cdot \mathbf{r}_u$$

where $I \in \mathbb{R}_{d \times d}$ is the identity matrix, and of course the reasoning is analogous with a fixed $\mathbf{x}_u$:

$$\mathbf{w}_i = \Big(X^T X + \lambda I \Big)^{-1} \cdot X^T \cdot \mathbf{r}_i$$

ALS algorithm is composed of the following steps:

1. Initialize all user latent vectors $\mathbf{X}$ and all item latent vectors $\mathbf{W}$ randomly.
2. Fix all item vectors $\mathbf{W}$ and solve for user vectors $\mathbf{X}$.
3. Fix all user vectors $\mathbf{X}$ and solve for item vectors $\mathbf{W}$.
4. Repeat steps 2 and 3 until convergence.

Convergence is guaranteed because, at each step, the loss function either decreases or remains unchanged.

In general, SGD is simpler and faster than ALS. However, ALS becomes preferable when parallelization is required, since each user and item vector can be updated independently, or when dealing with implicit feedback data, where the training set is dense and iterating over individual interactions, as SGD does, would be computationally impractical.

Matrix factorization offers flexibility in handling different aspects of data in collaborative filtering. Its core learning framework models the interactions between users and items that lead to varying rating values. However, much of the variation in ratings can be attributed to inherent biases of users or items, independent of their interactions. For instance, some users consistently rate items more generously, while some items tend to receive higher ratings overall.

Relying solely on user-item interactions to explain observed ratings may not be sufficiently accurate. To improve prediction, we separate latent factor modeling from bias modeling. For a given rating $r_{u,i}$, a first-order approximation of the associated bias $b_{u,i}$ is defined as:

$$b_{u,i} = \mu + b_u + b_i$$

where μ is the global average rating, b_u is the user bias (observed deviations of user u from the avg), and b_i is the item bias (observed deviations of item i from the avg).

For example, suppose we want a first-order estimate for user Joe's rating of the movie Titanic.

- $\mu = 3.7$: the global average rating over all movies
- $b_{\text{Titanic}} = 0.5$: Titanic is rated 0.5 stars above the average movie
- $b_{\text{Joe}} = -0.3$: Joe is a critical user who tends to give 0.3 fewer stars than average

Using the bias formula:

$$b_{\text{Joe,Titanic}} = \mu + b_{\text{Joe}} + b_{\text{Titanic}} = 3.7 - 0.3 + 0.5 = 3.9$$

So, Joe's estimated rating for *Titanic* is $\boxed{3.9}$.

To improve the accuracy of rating predictions, we extend the basic matrix factorization model by incorporating bias terms into the optimization process. The estimated rating $\hat{r}_{u,i}$ that user u gives to item i is now expressed as:

$$\hat{r}_{u,i} = \mathbf{x}_u^T \cdot \mathbf{w}_i + \mu + b_u + b_i$$

This prediction is composed of two main components:

- **Latent factor term** $\mathbf{x}_u^T \cdot \mathbf{w}_i$: This term models the interaction between the user and the item through their respective latent feature vectors $\mathbf{x}_u$ and $\mathbf{w}_i$.
- **Bias term** $\mu + b_u + b_i$: This component accounts for systematic effects not captured by the interaction term.

By combining them, the model can more accurately reflect both general trends and personalized user-item preferences. As a result, the original optimization problem becomes as follows:

$$X^*, W^* = \text{argmin}_{X,W} \left\{ \frac{1}{2} \sum_{(u,i)\in\mathcal{D}} \left[r_{u,i} - \left(\mathbf{x}_u^T \cdot \mathbf{w}_i + \mu + b_u + b_i \right) \right]^2 + \right.$$
$$\left. + \lambda \left(\sum_{u\in\mathcal{D}} \|\mathbf{x}_u\|^2 + \sum_{i\in\mathcal{D}} \|\mathbf{w}_i\|^2 + \sum_{u\in\mathcal{D}} b_u^2 + \sum_{i\in\mathcal{D}} b_i^2 \right) \right\}$$

and can still be solved using ALS.

5.2.3 Considerations

Collaborative filtering methods also suffer from several important limitations that affect their performance and applicability in real-world systems. The main issues are:

- **Cold Start:** When a new user or item enters the system, there is insufficient historical data to generate accurate recommendations.
- **Scalability:** With millions of users and items, CF systems may require significant computational resources to process data and produce recommendations in a timely manner.
- **Sparsity:** In most systems, users only rate a small fraction of available items, resulting in a sparse user-item matrix that makes it difficult to identify meaningful patterns.

5.3 Hybrid: Content-Based + Collaborative Filtering

Most modern recommender systems adopt a hybrid approach, combining the strengths of content-based and collaborative filtering techniques. These hybrid methods can be implemented in several ways, for instance, by merging the outputs of separate content-based and collaborative systems, by integrating content-based features into a collaborative framework (or vice versa), or by unifying both strategies into a single cohesive model.

Hybrid approaches have been shown to produce more accurate recommendations than using either method alone. Moreover, they help to address common issues in recommender systems, such as the cold start problem and sparsity in the user-item rating matrix. A well-known real-world example of a hybrid recommender system is Netflix, which successfully leverages both types of filtering to enhance user experience.

5.4 Evaluation Metrics

Recommendation systems are evaluated using both **offline** and **online** metrics. Offline evaluation includes measures such as RMSE, MAE, MAP@K, MAR@K, coverage, and personalization, while online evaluation relies on A/B testing to assess real-time performance through metrics like CTR, ROI, and other user interaction signals.

Root Mean Square Error (RMSE) is equivalent to the following formula:

$$\text{RMSE} = \frac{1}{|\mathcal{D}_{\text{test}}|}\sqrt{\sum_{(u,i)\in\mathcal{D}_{\text{test}}} \left(r_{u,i} - \hat{r}_{u,i}\right)^2}$$

it also presents some limitations since it may penalize methods that lack diversity in recommendations or that perform well on high ratings but poorly on others.

Additionally, it does not consider the order of recommended items, which can impact user satisfaction.

In binary classification, where the model predicts whether a condition is present or not, we define:

$$\text{Precision} = \frac{TP}{TP + FP}, \quad \text{Recall} = \frac{TP}{TP + FN}$$

These concepts can be adapted to recommender systems by mapping binary classification terms to analogous concepts in recommendation, as shown in figure 31.

binary classifier	recommender system
# with condition (y = 1)	# of all possible relevant items for a user
# predicted positive (TP + FP)	# of recommended items
# correct positives (TP)	# of recommended items that are relavant

Figure 31: Precision & recall in recommender systems.

Precision measures the proportion of recommended items that are actually relevant, while **recall** measures the proportion of relevant items that have been successfully recommended. For recommender systems we can therefore define:

$$\text{P} = \frac{\#relevant\ item\ recommendations}{\#items\ recommended}, \quad \text{R} = \frac{\#relevant\ item\ recommendations}{\#items\ actually\ relevant}$$

Precision and Recall in their standard form do not account for the ordering of recommendations. To address this, we use P@k and R@k , which evaluate performance by considering only the top-k recommendations, progressively measuring how many of the top-k items are relevant to the user, as shown in the following example.

$k = 3$

$P@3 = \frac{1}{3}$

Rank	Product Recommended	Result
1	Credit card	Correct positive
2	Christmas Fund	False positive
3	Debit Card	False positive
4	Auto loan	False positive
5	HELOC	Correct Positive
6	College Fund	Correct positive
7	Personal loan	False positive

$k = 6$

$P@6 = \frac{3}{6}$

Rank	Product Recommended	Result
1	Credit card	Correct positive
2	Christmas Fund	False positive
3	Debit Card	False positive
4	Auto loan	False positive
5	HELOC	Correct Positive
6	College Fund	Correct positive
7	Personal loan	False positive

Figure 32: P@k example.

Suppose our recommender system is required to return a ranked list of N recommended items, and let $|Rel|$ denote the number of actually relevant items for the user. We define the **Average Precision** at N (AP@N) as the average of the precision values computed at the ranks where relevant items appear. Formally, it is given by:

$$AP@N = \frac{1}{|Rel|} \sum_{k=1}^{N} P@k \cdot 1_{Rel}(k)$$

where $1_{Rel}(k)$ is an indicator function:

$$1_{Rel}(k) = \begin{cases} 1 & \text{if item at position } k \text{ is relevant} \\ 0 & \text{otherwise} \end{cases}$$

This new metric rewards methods that rank relevant items higher in the recommendation list, and is particularly useful when the order of results matters.

Since AP@N is is computed for a single data point (i.e., user) we also define the **Mean Average Precision** (MAP) as follows:

$$MAP@N = \frac{1}{|\mathcal{U}|} \sum_{u=1}^{|\mathcal{U}|} AP@N(u) = \frac{1}{|\mathcal{U}|} \sum_{u=1}^{|\mathcal{U}|} \frac{1}{|\,\mathrm{Rel}\,|} \sum_{k=1}^{N} P@k(u) \times 1_{\mathrm{Rel}}(k, u)$$

it provides a single-figure measure of recommendation quality by averaging the precision across multiple users, taking into account the rank of each relevant item.

A further metric is **personalization**, which measures how unique the recommendations are across different users by computing the dissimilarity between their recommendation lists, typically using 1-cosine similarity. A higher personalization score indicates that the system offers more tailored and diverse recommendations to each user.

Finally, let's suppose we have three users who are recommended the following lists of items:

$$u_1 = [A, B, C, D], \quad u_2 = [A, B, C, E], \quad u_3 = [A, B, F, G]$$

To evaluate how similar these recommendation lists are, we can represent each list as a binary vector over the full set of unique items {A, B, C, D, E, F}:

	A	B	C	D	E	F	G
u_1	1	1	1	1	0	0	0
u_2	1	1	1	0	1	0	0
u_3	1	1	0	0	0	1	1

Figure 33: Recommendation lists as binary vectors.

We then compute the **cosine similarity** between each pair of users' vectors. This results in a 3×3 similarity matrix M, as shown in figure 34.

$$M_{i,j} = \cos(u_i, u_j) = \frac{u_i \cdot u_j}{\|u_i\| \, \|u_j\|}$$

	u_1	u_2	u_3
u_1	1	0.75	0.58
u_2	0.75	1	0.58
u_3	0.58	0.58	1

-0.64

Figure 34: similarity matrix.

Since the matrix is symmetric and has 1s on the diagonal (each user is perfectly similar to themselves), we focus on the upper triangle of the matrix (excluding the diagonal). The resulting average reflects the overall similarity in recommendations across users. A lower value of the average indicates a higher level of **personalization** (personalization=1-AvgSim), meaning the system provides more distinct and individualized recommendations.

6 Link Analysis

Suppose we want to find an effective method to measure the **trustworthiness** of a page within the Web graph. This can be achieved by assigning a score that reflects the importance of a node in the graph. **Link analysis** seeks to compute such a score based solely on the structural properties of the graph, leveraging the fact that the Web is a typical example of a scale-free network.

A **scale-free network** is a type of network in which the distribution of node degrees follows a power law, meaning a few nodes (hubs) have many connections, while most have very few.

There are several link analysis approaches to compute web page importance:

- **PageRank**
- **Hubs and Authorities (HITS)**
- **Personalized PageRank**
- **Web Spam Detection**

6.1 PageRank

PageRank assigns a numerical score to each web page with the purpose of indicating its relative importance within the whole collection. It is based on 2 intuitions:

- The more incoming links a web page has, the more important it is. In other words each link from a web page w to a web page v can be considered as a vote by w to v.
- Links from important web pages should count more.

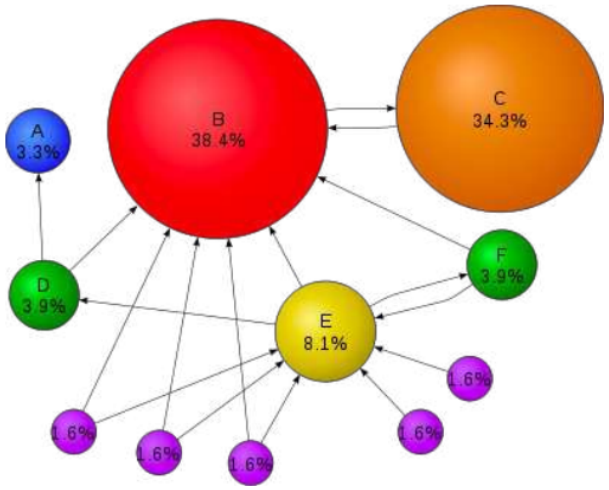


Figure 35: PageRank example.

For example, in figure 35 B has a high score since many nodes point to it, but C also has a high score even though it has only one incoming link, and that's because the link comes from the important node B.

PageRank can be described through the following objects:

- $G = (V, E)$ is the web graph, and $|V| = N$ is the number of nodes (pages) in V,
- $O_v = \{w \in V : (v, w) \in E\}$ is the set of pages linked by v,
- $|O_v| = o_v$ is called the out-degree of node v,
- $I_v = \{w \in V : (w, v) \in E\}$ is the set of pages linked to v,
- $|I_v| = i_v$ is called the in-degree of node v.

As already discussed in the example, each vote to a page is proportional to the importance of the source page. More precisely, if a page w has importance r_w and out-degree o_w, each out-link will distribute an equal proportion of its importance, thus r_w/o_w. On the other hand, the importance of each page can be computed as the sum of the votes of all its incoming links.

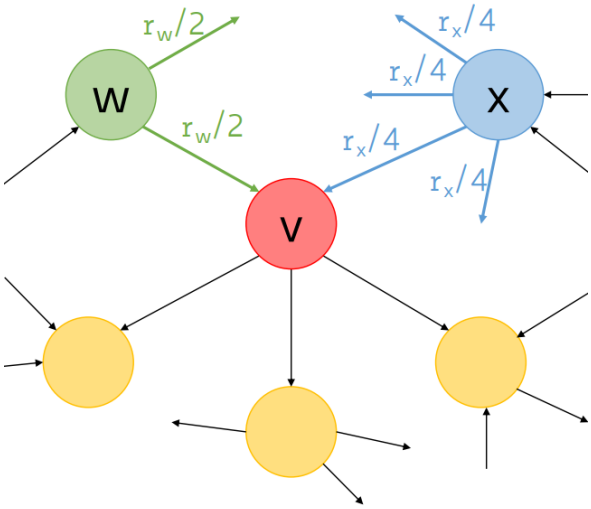


Figure 36: Importance distribution.

For example, in figure 36 $r_v = r_w/2 + r_x/4$, in the more general case we have:

$$r_v = \sum_{u \in I_v} \frac{r_u}{o_u}$$

PageRank can also be interpreted from two main perspectives:

- **Linear Algebra**
- **Probability**

6.1.1 Linear Algebra Perspective

In linear algebra, as shown in figure 37 importance can be determined through the so-called **flow equations**. Moreover, systems of flow equations need to be normalized by adding a constraint; otherwise, the system would accept an infinite number of solutions. This perspective may work for very small systems of linear equations, but in the case of web pages, we might have hundreds of billions of equations, which is impracticable. Thus, we need a new formulation.

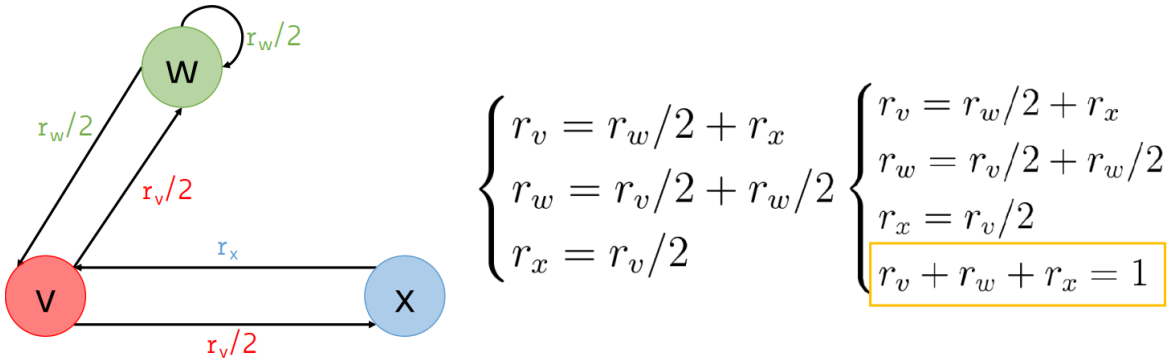


Figure 37: Flow equations system example.

It is possible to represent the web graph as a **column-stochastic matrix** M of size $N \times N$, where $N = |V|$. In this matrix, the entry $M[u, w]$ is equal to zero if node w does not link to node u, and equal to $\frac{1}{o_w}$ otherwise. A column-stochastic vector is defined as a vector of non-negative entries whose elements sum to 1. Naturally, assuming that each column sums to 1 implies that every node has at least one outgoing link.

Let's also define $\mathbf{r}$ as the **rank vector** of dimension N, in which each entry r_v represents the rank score (the importance) of page v, and the sum of all entries is equal 1:

$$\mathbf{r} = [r_1, \ldots, r_N]^T \quad \text{where} \quad r_v = \sum_{u \in I_v} \frac{r_u}{o_u} \quad \text{and} \quad \sum_{v=1}^{N} r_v = 1$$

then it can be written in a matrix formulation $\mathbf{r} = M\mathbf{r}$, since each $r_v = m_v^T \cdot \mathbf{r}$.

Intuitively, looking at column u of the matrix M allows us to identify the outgoing links from page u, along with the fraction of its rank passed to each linked page. On the other hand, examining row v reveals the incoming links to page v, and how much rank it receives from each linking page.

It is worth noting that $\mathbf{r}$ is the eigenvector of matrix M when λ is equal 1:

$$A\mathbf{x} = \lambda\mathbf{x} := M\mathbf{r} = \mathbf{r} \quad \text{where} \quad \lambda = 1 \ .$$

For a given eigenvalue, all corresponding eigenvectors are scalar multiples of each other, so any of them can represent the PageRank vector $\mathbf{r}$. To reflect only relative importance, we choose the probabilistic eigenvector, the one whose entries sum to 1, corresponding to the eigenvalue $\lambda = 1$.

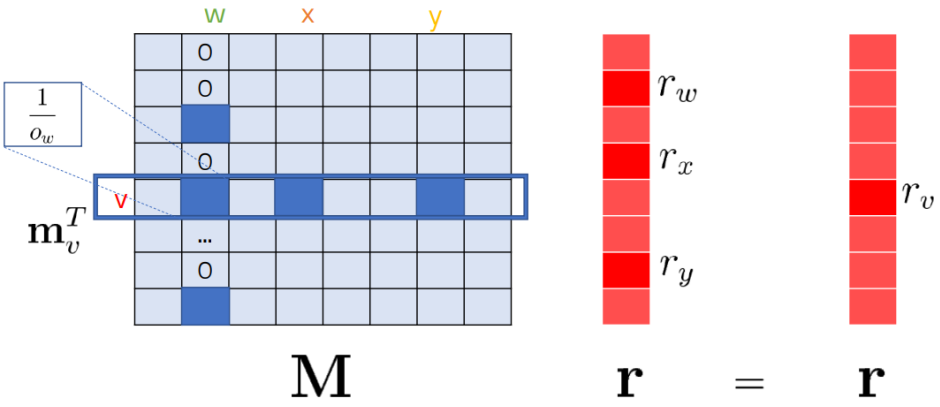


Figure 38: Matrix formulation of PageRank.

From linear algebra, we know that any stochastic matrix M has a largest eigenvalue $\lambda = 1$, and the PageRank vector $\mathbf{r} = \mathbf{r}^*$ is the principal eigenvector associated with it. Also note that so far, we've assumed M is column-stochastic, but this doesn't always hold in the case of a general web graph.

So starting from flow equations, we reformulated the system using linear algebra, and reduced the problem to finding the eigenvector of the matrix M. Such a problem can be solved using the **power iteration method**.

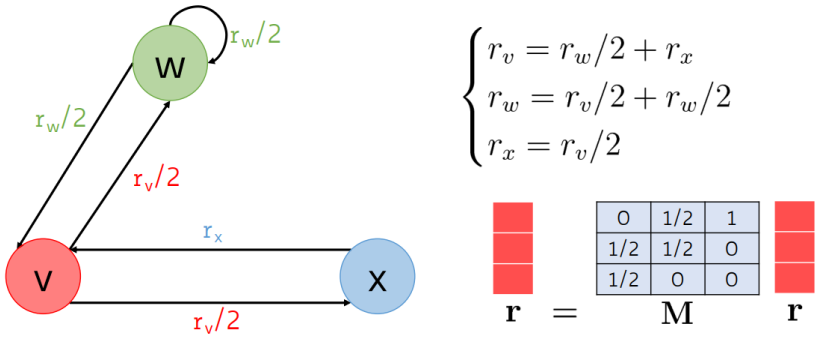


Figure 39: From flow equations to matrix.

Power Iteration Method: It begins by assuming that all pages have the same rank score, uniformly distributed across the N pages:

$$\text{init: } t = 0;\ \mathbf{r}(t) = (\frac{1}{N}, \frac{1}{N}, \ldots, \frac{1}{N})^T$$

then the rank vector $\mathbf{r}$ is updated until convergence:

$$\text{repeat: } \mathbf{r}(t+1) = M\mathbf{r}(t)$$
$$\text{until: } \delta(\mathbf{r}(t+1), \mathbf{r}(t)) < \epsilon$$
$$\text{where } \epsilon > 0 \text{ and } \begin{cases} \delta(\mathbf{r}(t+1), \mathbf{r}(t)) = |\mathbf{r}(t+1) - \mathbf{r}(t)| \text{ or} \\ \delta(\mathbf{r}(t+1), \mathbf{r}(t)) = \|\mathbf{r}(t+1) - \mathbf{r}(t)\| \end{cases}$$

this allows us to solve the system of equations in a non-explicit way.

6.1.2 Probabilistic Perspective

Imagine a random **surfer** navigating through the web, moving from one page to another. At the initial time, $t = 0$, the surfer has an equal chance of starting on any web page in the graph ($\frac{1}{N}$ for N web pages).

At any given time t, the surfer is located on some web page w. To decide the next move, the surfer randomly selects one of the outgoing links from page w, choosing uniformly among them. The surfer then follows that link and lands on a new page v. This process continues indefinitely and forms what is known as a **random walk** over the web graph.

We can represent this behavior using the matrix M of size $N \times N$, in which each entry $m_{v,w}$ can be seen as the probability of transitioning from page w to page v. In other words, the column w of the matrix M contains the probability distribution over the pages that can be reached from page w.

The matrix M defines a **Markov chain** over the finite set of web pages, treating each page as a state in the system.

Here is a formal definition of the random walk interpretation. Let X be a discrete random variable that can take on $|V| = N$ possible values, where each value corresponds to a unique web page. The expression $X = w$ indicates that a random surfer is currently on web page w.

We define an N-dimensional stochastic (probability) vector associated with X as:

$$\mathbf{p} \in \mathbb{R}^N = \begin{pmatrix} P(X = 1) \\ \vdots \\ P(X = w) \\ \vdots \\ P(X = N) \end{pmatrix}$$

This vector represents the probability distribution over all web pages.

To model the surfer's position over time, we define a time-dependent stochastic vector $\mathbf{p}(t) \in \mathbb{R}^N$:

$$\mathbf{p}(t) = \begin{pmatrix} P(X_t = 1) \\ \vdots \\ P(X_t = w) \\ \vdots \\ P(X_t = N) \end{pmatrix}$$

The evolution of $\mathbf{p}(t)$ over time is governed by the structure of the web graph and forms the foundation of the PageRank algorithm.

Markov property states that the transition probability to the next state depends only on the current state and not on the sequence of previous states. Formally, the

Markov property is given by:

$$P(X_{t+1} = v \mid X_1 = x_1, \ldots, X_t = x_t) = P(X_{t+1} = v \mid X_t = x_t)$$

Thus, the probability of the surfer being on page v at time $t+1$ depends only on the page they were on at time t, and not on how they got there.

Suppose we want to estimate the probability that the surfer is on page v at time $t+1$, and assume that page v has exactly three incoming links from w, x, and y as shown in figure 40.

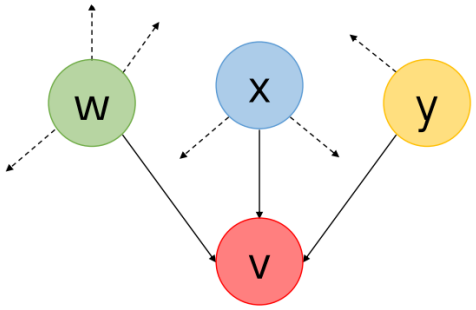


Figure 40: Walk interpretation example.

Then we can write:

$$P(X_{t+1} = v) = P(X_t = w, Z_w = v) + P(X_t = x, Z_x = v) + P(X_t = y, Z_y = v)$$

where Z_u represents the page that the surfer jumps to from page u, chosen uniformly at random from the outgoing links of u:

$$Z_u \sim \text{Uniform}(1, O_u)$$

where O_u is the number of outgoing links from page u.

This naturally leads to the update equation for the probability vector:

$$\mathbf{p}(t+1) = M\mathbf{p}(t)$$

which is structurally identical to the PageRank update rule:

$$\mathbf{r}(t+1) = M\mathbf{r}(t)$$

Solving this recursive equation for the steady state is therefore equivalent to computing the PageRank vector. This vector gives the long-term probabilities of the random surfer being on each page.

Initially, at time $t = 0$, the random surfer is equally likely to be on any of the N web pages:

$$\mathbf{p}(0) = \begin{pmatrix} \frac{1}{N} \\ \vdots \\ \frac{1}{N} \end{pmatrix}$$

After one time step, the probability distribution over the pages is updated as:

$$\mathbf{p}(1) = M\mathbf{p}(0)$$

where M is the transition matrix derived from the link structure of the web. The v-th entry of $\mathbf{p}(1)$ gives the probability that the surfer is on page v after one move.

More generally, after t steps, the probability distribution is given by:

$$\mathbf{p}(t) = M\mathbf{p}(t-1) = M \times M \times ... \times M\mathbf{p}(0) = M^t\mathbf{p}(0)$$

this sequence converges to a steady state, where the stochastic vector doesn't change anymore. Such a stationary distribution is denoted with $\mathbf{p}^*$.

6.1.3 Conclusions on Both Perspectives

The steady-state vector $\mathbf{p}^*$ is the same as the PageRank vector $\mathbf{r}^*$ obtained via the power iteration method. But how can we be sure that this vector:

- exists,
- is unique,
- will converge.

These properties are guaranteed under certain conditions on the matrix M. Specifically, if M is a column-stochastic matrix with all positive entries, then:

- $\lambda = 1$ is an eigenvalue of M, and it has **multiplicity one**,
- $\lambda = 1$ is the **largest eigenvalue** (in absolute value),
- There exists a **unique** right eigenvector $\mathbf{r}^*$ associated with $\lambda = 1$, and it can be scaled such that the sum of its entries is equal to 1.

From the probabilistic perspective, if M is a column-stochastic matrix with all positive entries, then for any initial distribution $\mathbf{p}(0)$:

$$\mathbf{p}(t) = M^t\mathbf{p}(0) \quad \text{converges to} \quad \mathbf{p}^* \quad \text{as} \quad t \longrightarrow \infty$$

These properties follow from the **Perron–Frobenius theorem**, which applies to positive stochastic matrices. As a result, the power iteration method converges to this unique stationary distribution, the PageRank vector.

Unfortunately, we cannot directly apply the Perron–Frobenius theorem to the original matrix M as defined, because it is not guaranteed to be positive, some entries may be zero. This reflects the irregular and heterogeneous structure of the Web graph. This is precisely where the Google PageRank algorithm comes in; it modifies the original matrix to ensure the necessary properties (such as positivity), allowing the convergence to a unique PageRank vector.

6.1.4 Google's PageRank

There are two major issues with the original PageRank formulation:

- **Dead ends**: Pages with no outgoing links (**dangling nodes**)cause the random surfer to get "stuck", effectively removing probability mass from the system and leading to rank leakage.

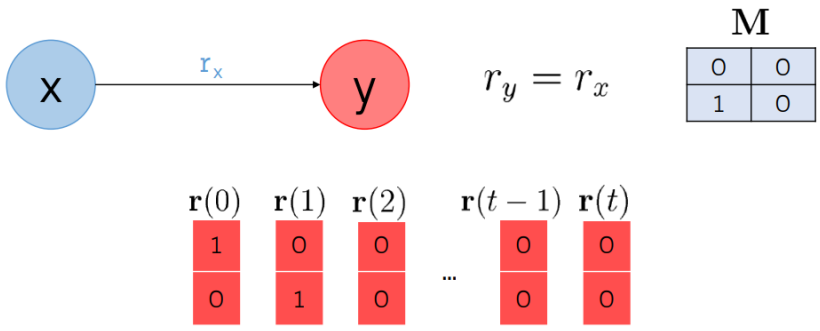


Figure 41: The PageRank vector vanishes to 0.

In other words, M is not column stochastic as some nodes have no outlinks.

- **Spider traps**: Groups of pages that only link to each other (but not to the rest of the graph) can absorb the entire PageRank.

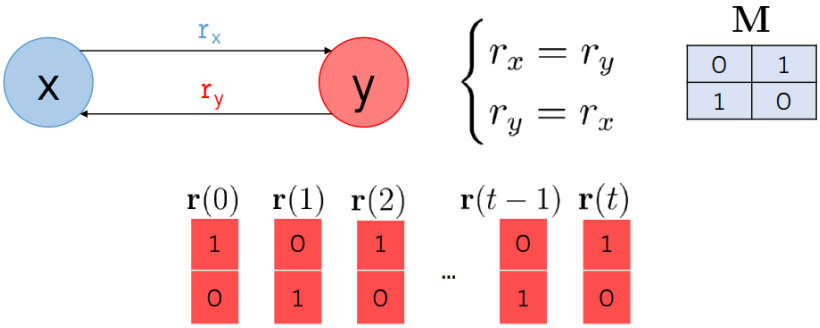


Figure 42: The vector keeps alternating without convergence.

In other words, even if M is stochastic, it is not strictly positive.

Deal with Dangling Nodes: When z is a dangling node, its column in M is exclusively composed of 0s. If we apply simplified PageRank to M the rank vector $\mathbf{r}$ will eventually vanish to 0. The solution to this problem is to create artificial links from any dangling node to any other node:

$$\begin{bmatrix} w & x & y & z \\ \hline 0 & 0 & 1/2 & 0 \\ 1/3 & 0 & 0 & 0 \\ 1/3 & 0 & 0 & 0 \\ 1/3 & 1 & 1/2 & 0 \end{bmatrix} \longrightarrow \begin{bmatrix} w & x & y & z \\ \hline 0 & 0 & 1/2 & 1/4 \\ 1/3 & 0 & 0 & 1/4 \\ 1/3 & 0 & 0 & 1/4 \\ 1/3 & 1 & 1/2 & 1/4 \end{bmatrix}$$

This adjustment is justified by modeling the behavior of a web surfer who, upon reaching a page with no outgoing links, jumps to a randomly selected page among the N available pages, with equal probability. In such a scenario M becomes the following matrix M':

$$M'_{N\times N} := \quad m'_{v,w} = \begin{cases} \frac{1}{O_w} & \text{if} \quad v \in O_w \\ \frac{1}{N} & \text{if} \quad \sum_{v=1}^{N} m_{v,w} = 0 \\ 0 & \text{otherwise} \end{cases}$$

This transformation allows M' to be column stochastic.

Deal with Spider Traps: Assume that M is already column stochastic, as in the case of figure 43. If we applied simplified PageRank to M, some entries of the rank

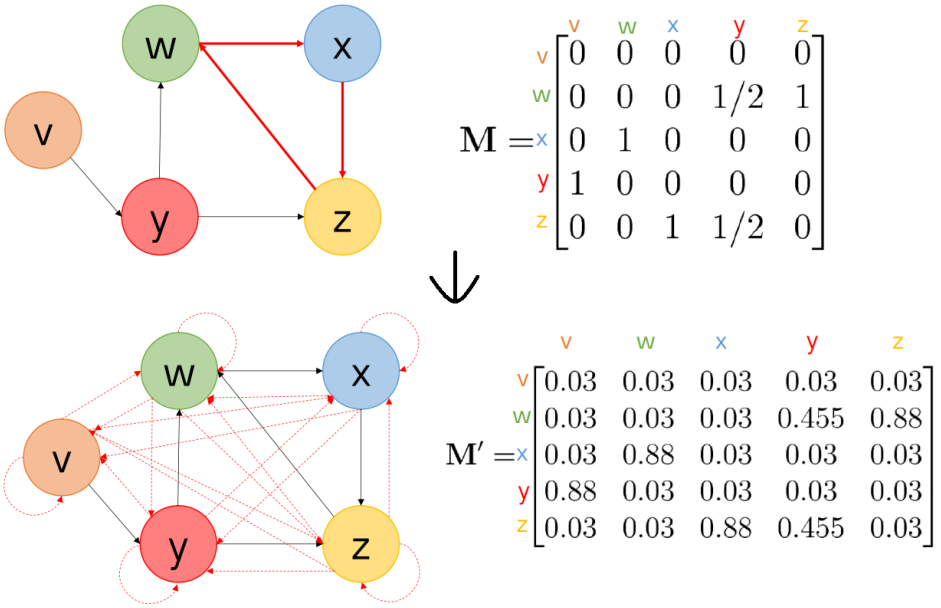


Figure 43: Example of spider trap solution.

vector $\mathbf{r}$ will eventually drop to 0, as we get stuck in w, x, or z. Again, the solution is to create artificial links from each node to every other node, and follow each of them with probability $(1-d)/N$. In this way, on each page w the surfer will either follow one of its outgoing links with probability d (known as **damping factor**) or jump to another page with probability $1-d$. In the original Google formulation $d = 0.85$.

Thus, the actual matrix of the Google's PageRank formulation is the following matrix G:

$$G = dM' + \frac{1-d}{N}\begin{bmatrix} 1 & 1 & \dots & 1 \\ 1 & 1 & \dots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \dots & 1 \end{bmatrix}$$

While M' ensures that the matrix is column stochastic, the matrix G guarantees that all entries are strictly positive. Ultimately ensuring convergence to $\mathbf{r}^*$.

Compute PageRank: At this point, computing $\mathbf{r}(t+1)$ is easy if we have enough memory to store G, $\mathbf{r}(t+1)$, and $\mathbf{r}(t)$. Unfortunately, G represents a fully-connected graph with a huge number of nodes (dense matrix).

Assuming the number of web pages in the graph is $N = 10^9$, the matrix G will have $N^2 = 10^{18}$ entries. If each entry is stored as a 32-bit integer (4 bytes), storing the entire matrix G would require

$$4 \times 10^{18} \text{ bytes} = 4 \text{ EB (exabytes)}.$$

In addition to the matrix, we need to store the PageRank vectors $\mathbf{r}(t)$ and $\mathbf{r}(t+1)$, each containing $N = 10^9$ entries. At 4 bytes per entry, each vector requires 4 gigabytes (GB), so together they require 8 GB. It is also important to note that the real Web contains far more than 10^9 pages, making it infeasible to store and process the full matrix G directly with current technology. The idea is to re-arrange the equation:

$$\text{starting from} \quad \mathbf{r} = \mathbf{G}\mathbf{r} \quad \text{and} \quad \mathbf{G}_{v,w} = dM'_{v,w} + \frac{1-d}{N}$$

$$\text{we try to re-arrange} \quad r_v = \sum_{w=1}^{N} G_{v,w} \times r_w \rightarrow r_v = \sum_{w=1}^{N} \left(dM'_{v,w} + \frac{1-d}{N} \right) \times r_w$$

$$\rightarrow r_v = \sum_{w=1}^{N} dM'_{v,w} \times r_w + \sum_{w=1}^{N} \frac{1-d}{N} \times r_w \rightarrow r_v = \sum_{w=1}^{N} dM'_{v,w} \times r_w + \frac{1-d}{N} \sum_{w=1}^{N} r_w$$

$$\text{but} \quad \sum_{w=1}^{N} r_w = 1 \quad \text{thus} \quad r_v = \sum_{w=1}^{N} dM'_{v,w} \times r_w + \frac{1-d}{N}$$

$$\rightarrow \mathbf{r} = dM'\mathbf{r} + \begin{bmatrix} \frac{1-d}{N} \\ \vdots \\ \frac{1-d}{N} \end{bmatrix}$$

Since each web page has approximately 10 outgoing links, the matrix M' contains about 10^{10} nonzero entries, which reduces the memory needed to store it by a factor of 10^8 compared to the full matrix G; therefore, it is feasible to work with M' instead of G.

Thus, at each iteration we can compute PageRank vector following 2 steps:

1. $\mathbf{r}(t+1) = dM'\mathbf{r}(t)$
2. $\mathbf{r}(t+1) = \mathbf{r}(t+1) + \begin{bmatrix} \frac{1-d}{N} \\ \vdots \\ \frac{1-d}{N} \end{bmatrix}$

In other words, we add the constant $\frac{1-d}{N}$ to each component of $\mathbf{r}(t+1)$ only after the first step, which allows us to avoid computations involving a dense matrix.