Four Paradigms in Data Mining

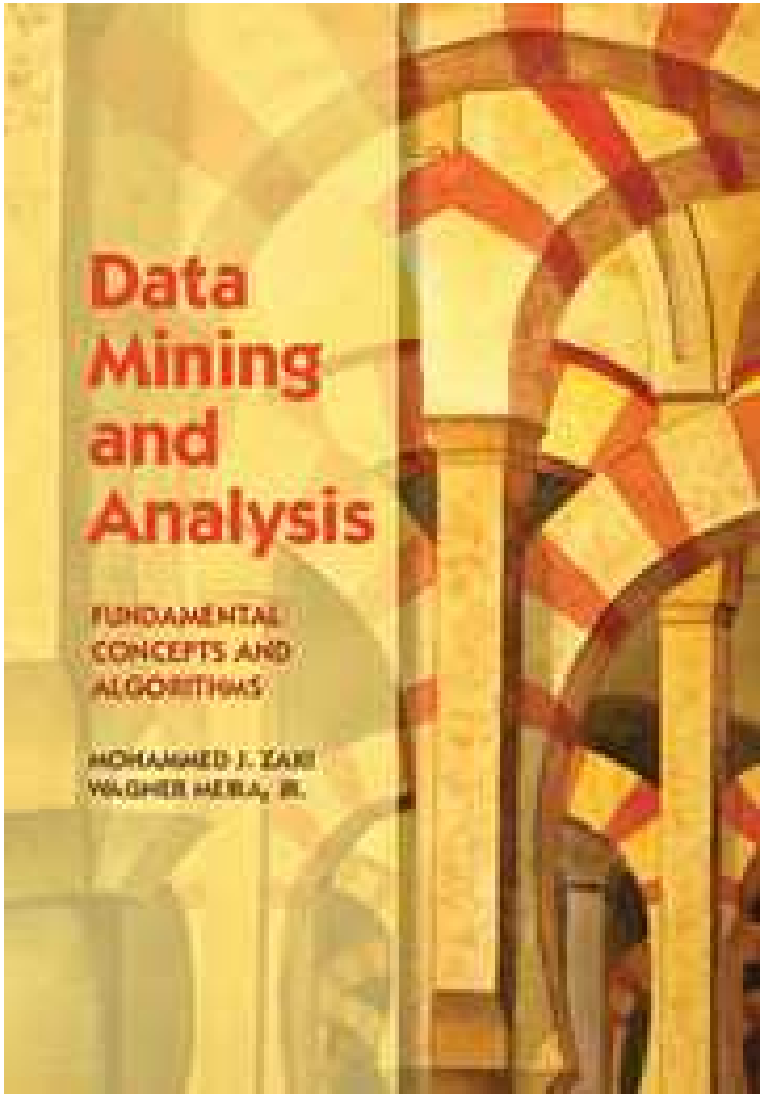
dataminingbook.info

Wagner Meira Jr.[1]

[1]Department of Computer Science
Universidade Federal de Minas Gerais, Belo Horizonte, Brazil

October 13, 2015

Book



Cambridge Press, 2014, 624 pages.

PDF freely available at
http://dataminingbook.info/

Data

- Tabular
  - categorical
  - numeric
- Text
- Graphs
- Sound
- Image
- Video

Data

Challenges

- Storage
- Accessing
- Engineering
  - Integration
  - Cleaning
  - Transformation
- Visualization

Data Mining

Concept

Automatic extraction of knowledge or patterns that are interesting (novel, useful, implicit, etc.) from large volumes of data.

Tasks

- Data engineering
- Characterization
- Prediction

Data Mining Models

Concept

A model aims to represent the nature or reality from a specific perspective. A model is an artificial construction where all extraneous details have been removed or abstracted, while keeping the key features necessary for analysis and understanding.

Task

Among all possible sets of entities, which ones are the most frequent? Or better, determine the sets of items that co-occur in a database more frequently than a given threshold.

Application Scenario

Market-basket problem: Given that a customer purchased items in set A, what are the most likely items to be purchased in the future?

Task

Given a similarity criterion, what is the entity partition that groups together the most similar entities?

Application Scenario

Customer segmentation: Partition a customer base into groups of similar customers, supporting different policies and strategies for each group.

Task

Given some knowledge about a domain, including classes or categories of entities, and a sample whose class is unknown, predict the class of the latter based on the existing knowledge.

Application Scenario

Credit scoring: A bank needs to decide whether it will loan money to a given person. It may use past experience with other persons who present a similar profile to decide whether or not it is worth giving the loan.

Data Mining

Paradigms

- **Combinatorial**
- Probabilistic
- Algebraic
- Graph-based

Domain

Models partition (or select) entities based on their attributes and their combinations. Search space is discrete and finite, although potentially very large.

Task

Determine the best model according to a quality metric.

Strategies

- Pruning exhaustive search
- Heuristic approximation

Data Matrix

Data can often be represented or abstracted as an $n \times d$ *data matrix*, with n rows and d columns, given as

$$\mathbf{D} = \begin{array}{c|cccc} & X_1 & X_2 & \cdots & X_d \\ \hline \mathbf{x}_1 & x_{11} & x_{12} & \cdots & x_{1d} \\ \mathbf{x}_2 & x_{21} & x_{22} & \cdots & x_{2d} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \mathbf{x}_n & x_{n1} & x_{n2} & \cdots & x_{nd} \end{array}$$

- **Rows**: Also called *instances*, *examples*, *records*, *transactions*, *objects*, *points*, *feature-vectors*, etc. Given as a d-tuple

$$\mathbf{x}_i = (x_{i1}, x_{i2}, \ldots, x_{id})$$

- **Columns**: Also called *attributes*, *properties*, *features*, *dimensions*, *variables*, *fields*, etc. Given as an n-tuple

$$X_j = (x_{1j}, x_{2j}, \ldots, x_{nj})$$

Iris Dataset Extract

	Sepal length X_1	**Sepal width** X_2	**Petal length** X_3	**Petal width** X_4	**Class** X_5
$\mathbf{x}_1$	5.9	3.0	4.2	1.5	Iris-versicolor
$\mathbf{x}_2$	6.9	3.1	4.9	1.5	Iris-versicolor
$\mathbf{x}_3$	6.6	2.9	4.6	1.3	Iris-versicolor
$\mathbf{x}_4$	4.6	3.2	1.4	0.2	Iris-setosa
$\mathbf{x}_5$	6.0	2.2	4.0	1.0	Iris-versicolor
$\mathbf{x}_6$	4.7	3.2	1.3	0.2	Iris-setosa
$\mathbf{x}_7$	6.5	3.0	5.8	2.2	Iris-virginica
$\mathbf{x}_8$	5.8	2.7	5.1	1.9	Iris-virginica
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\mathbf{x}_{149}$	7.7	3.8	6.7	2.2	Iris-virginica
$\mathbf{x}_{150}$	5.1	3.4	1.5	0.2	Iris-setosa

Attributes

Attributes may be classified into two main types

- **Numeric Attributes**: real-valued or integer-valued domain
  - *Interval-scaled*: only differences are meaningful
    e.g., `temperature`
  - *Ratio-scaled*: differences and ratios are meaningful
    e..g, `Age`
- **Categorical Attributes**: set-valued domain composed of a set of symbols
  - *Nominal*: only equality is meaningful
    e.g., domain(`Sex`) = { `M, F`}
  - *Ordinal*: both equality (are two values the same?) and inequality (is one value less than another?) are meaningful
    e.g., domain(`Education`) = { `High School`, `BS`, `MS`, `PhD`}

- **Frequent Itemset Mining**
- k-Means
- DBScan
- Decision trees

Frequent Itemset Mining

In many applications one is interested in how often two or more objects of interest co-occur, the so-called *itemsets*.

The prototypical application was *market basket analysis*, that is, to mine the sets of items that are frequently bought together at a supermarket by analyzing the customer shopping carts (the so-called "market baskets").

Frequent itemset mining is a basic exploratory mining task, since the since the basic operation is to find the co-occurrence, which gives an estimate for the joint probability mass function.

Once we mine the frequent sets, they allow us to extract *association rules* among the itemsets, where we make some statement about how likely are two sets of items to co-occur or to conditionally occur.

Frequent Itemsets: Terminology

Itemsets: Let $\mathcal{I} = \{x_1, x_2, \ldots, x_m\}$ be a set of elements called *items*. A set $X \subseteq \mathcal{I}$ is called an *itemset*. An itemset of cardinality (or size) k is called a k-itemset. Further, we denote by $\mathcal{I}^{(k)}$ the set of all k-itemsets, that is, subsets of $\mathcal{I}$ with size k.

Tidsets: Let $\mathcal{T} = \{t_1, t_2, \ldots, t_n\}$ be another set of elements called transaction identifiers or *tids*. A set $T \subseteq \mathcal{T}$ is called a *tidset*. Itemsets and tidsets are kept sorted in lexicographic order.

Transactions: A *transaction* is a tuple of the form $\langle t, X \rangle$, where $t \in \mathcal{T}$ is a unique transaction identifier, and X is an itemset.

Database: A binary database $\mathbf{D}$ is a binary relation on the set of tids and items, that is, $\mathbf{D} \subseteq \mathcal{T} \times \mathcal{I}$. We say that tid $t \in \mathcal{T}$ *contains* item $x \in \mathcal{I}$ iff $(t, x) \in \mathbf{D}$. In other words, $(t, x) \in \mathbf{D}$ iff $x \in X$ in the tuple $\langle t, X \rangle$. We say that tid t *contains* itemset $X = \{x_1, x_2, \ldots, x_k\}$ iff $(t, x_i) \in \mathbf{D}$ for all $i = 1, 2, \ldots, k$.

Database Representation

Let 2^X denote the powerset of X, that is, the set of all subsets of X. Let $\mathbf{i} : 2^{\mathcal{T}} \rightarrow 2^{\mathcal{I}}$ be a function, defined as follows:

$$\mathbf{i}(T) = \{x \mid \forall t \in T,\ t \text{ contains } x\}$$

where $T \subseteq \mathcal{T}$, and $\mathbf{i}(T)$ is the set of items that are common to *all* the transactions in the tidset T. In particular, $\mathbf{i}(t)$ is the set of items contained in tid $t \in \mathcal{T}$.

Let $\mathbf{t} : 2^{\mathcal{I}} \rightarrow 2^{\mathcal{T}}$ be a function, defined as follows:

$$\mathbf{t}(X) = \{t \mid t \in \mathcal{T} \text{ and } t \text{ contains } X\} \tag{1}$$

where $X \subseteq \mathcal{I}$, and $\mathbf{t}(X)$ is the set of tids that contain *all* the items in the itemset X. In particular, $\mathbf{t}(x)$ is the set of tids that contain the single item $x \in \mathcal{I}$.

The binary database $\mathbf{D}$ can be represented as a *horizontal* or *transaction database* consisting of tuples of the form $\langle t, \mathbf{i}(t) \rangle$, with $t \in \mathcal{T}$.

The binary database $\mathbf{D}$ can also be represented as a *vertical* or *tidset database* containing a collection of tuples of the form $\langle x, \mathbf{t}(x) \rangle$, with $x \in \mathcal{I}$.

Binary Database: Transaction and Vertical Format

D	A	B	C	D	E
1	1	1	0	1	1
2	0	1	1	0	1
3	1	1	0	1	1
4	1	1	1	0	1
5	1	1	1	1	1
6	0	1	1	1	0

Binary Database

t	$\mathbf{i}(t)$
1	$ABDE$
2	BCE
3	$ABDE$
4	$ABCE$
5	$ABCDE$
6	BCD

Transaction Database

$\mathbf{t}(x)$				
A	B	C	D	E
1	1	2	1	1
3	2	4	3	2
4	3	5	5	3
5	4	6	6	4
	5			5
	6			

Vertical Database

This dataset **D** has 5 items, $\mathcal{I} = \{A, B, C, D, E\}$ and 6 tids $\mathcal{T} = \{1, 2, 3, 4, 5, 6\}$.

The the first transaction is $\langle 1, \{A, B, D, E\}\rangle$, where we omit item C since $(1, C) \notin \mathbf{D}$. Henceforth, for convenience, we drop the set notation for itemsets and tidsets. Thus, we write $\langle 1, \{A, B, D, E\}\rangle$ as $\langle 1, ABDE\rangle$.

Support and Frequent Itemsets

The *support* of an itemset X in a dataset $\mathbf{D}$, denoted $sup(X)$, is the number of transactions in $\mathbf{D}$ that contain X:

$$sup(X) = \left|\{t \mid \langle t, \mathbf{i}(t)\rangle \in \mathbf{D} \text{ and } X \subseteq \mathbf{i}(t)\}\right| = |\mathbf{t}(X)|$$

The *relative support* of X is the fraction of transactions that contain X:

$$rsup(X) = \frac{sup(X)}{|\mathbf{D}|}$$

It is an estimate of the *joint probability* of the items comprising X.

An itemset X is said to be *frequent* in $\mathbf{D}$ if $sup(X) \geq minsup$, where $minsup$ is a user defined *minimum support threshold*.

The set $\mathcal{F}$ to denotes the set of all frequent itemsets, and $\mathcal{F}^{(k)}$ denotes the set of frequent k-itemsets.

Frequent Itemsets

Minimum support: $minsup = 3$

t	$\mathbf{i}(t)$
1	$ABDE$
2	BCE
3	$ABDE$
4	$ABCE$
5	$ABCDE$
6	BCD

Transaction Database

sup	itemsets
6	B
5	E, BE
4	$A, C, D, AB, AE, BC, BD, ABE$
3	$AD, CE, DE, ABD, ADE, BCE, BDE, ABDE$

Frequent Itemsets

The 19 frequent itemsets shown in the table comprise the set $\mathcal{F}$. The sets of all frequent k-itemsets are

$$\mathcal{F}^{(1)} = \{A, B, C, D, E\}$$
$$\mathcal{F}^{(2)} = \{AB, AD, AE, BC, BD, BE, CE, DE\}$$
$$\mathcal{F}^{(3)} = \{ABD, ABE, ADE, BCE, BDE\}$$
$$\mathcal{F}^{(4)} = \{ABDE\}$$

Association Rules

An *association rule* is an expression

$$X \xrightarrow{s,c} Y$$

where X and Y are itemsets and they are disjoint, that is, $X, Y \subseteq \mathcal{I}$, and $X \cap Y = \emptyset$. Let the itemset $X \cup Y$ be denoted as XY.

The *support* of the rule is the number of transactions in which both X and Y co-occur as subsets:

$$s = sup(X \longrightarrow Y) = |\mathbf{t}(XY)| = sup(XY)$$

The *relative support* of the rule is defined as the fraction of transactions where X and Y co-occur, and it provides an estimate of the joint probability of X and Y:

$$rsup(X \longrightarrow Y) = \frac{sup(XY)}{|\mathbf{D}|} = P(X \wedge Y)$$

The *confidence* of a rule is the conditional probability that a transaction contains Y given that it contains X:

$$c = conf(X \longrightarrow Y) = P(Y|X) = \frac{P(X \wedge Y)}{P(X)} = \frac{sup(XY)}{sup(X)}$$

Itemset Mining Algorithms: Brute Force

The brute-force algorithm enumerates all the possible itemsets $X \subseteq \mathcal{I}$, and for each such subset determines its support in the input dataset $\mathbf{D}$. The method comprises two main steps: (1) candidate generation and (2) support computation.

Candidate Generation: This step generates all the subsets of $\mathcal{I}$, which are called *candidates*, as each itemset is potentially a candidate frequent pattern. The candidate itemset search space is clearly exponential because there are $2^{|\mathcal{I}|}$ potentially frequent itemsets.

Support Computation: This step computes the support of each candidate pattern X and determines if it is frequent. For each transaction $\langle t, \mathbf{i}(t) \rangle$ in the database, we determine if X is a subset of $\mathbf{i}(t)$. If so, we increment the support of X.

Computational Complexity: Support computation takes time $O(|\mathcal{I}| \cdot |\mathbf{D}|)$ in the worst case, and because there are $O(2^{|\mathcal{I}|})$ possible candidates, the computational complexity of the brute-force method is $O(|\mathcal{I}| \cdot |\mathbf{D}| \cdot 2^{|\mathcal{I}|})$.

Brute Force Algorithm

BRUTEFORCE (D, $\mathcal{I}$, *minsup*):

```
1 F ← ∅ // set of frequent itemsets
2 foreach X ⊆ I do
3     sup(X) ← COMPUTESUPPORT (X, D)
4     if sup(X) ≥ minsup then
5         F ← F ∪ {(X, sup(X))}
6 return F
```

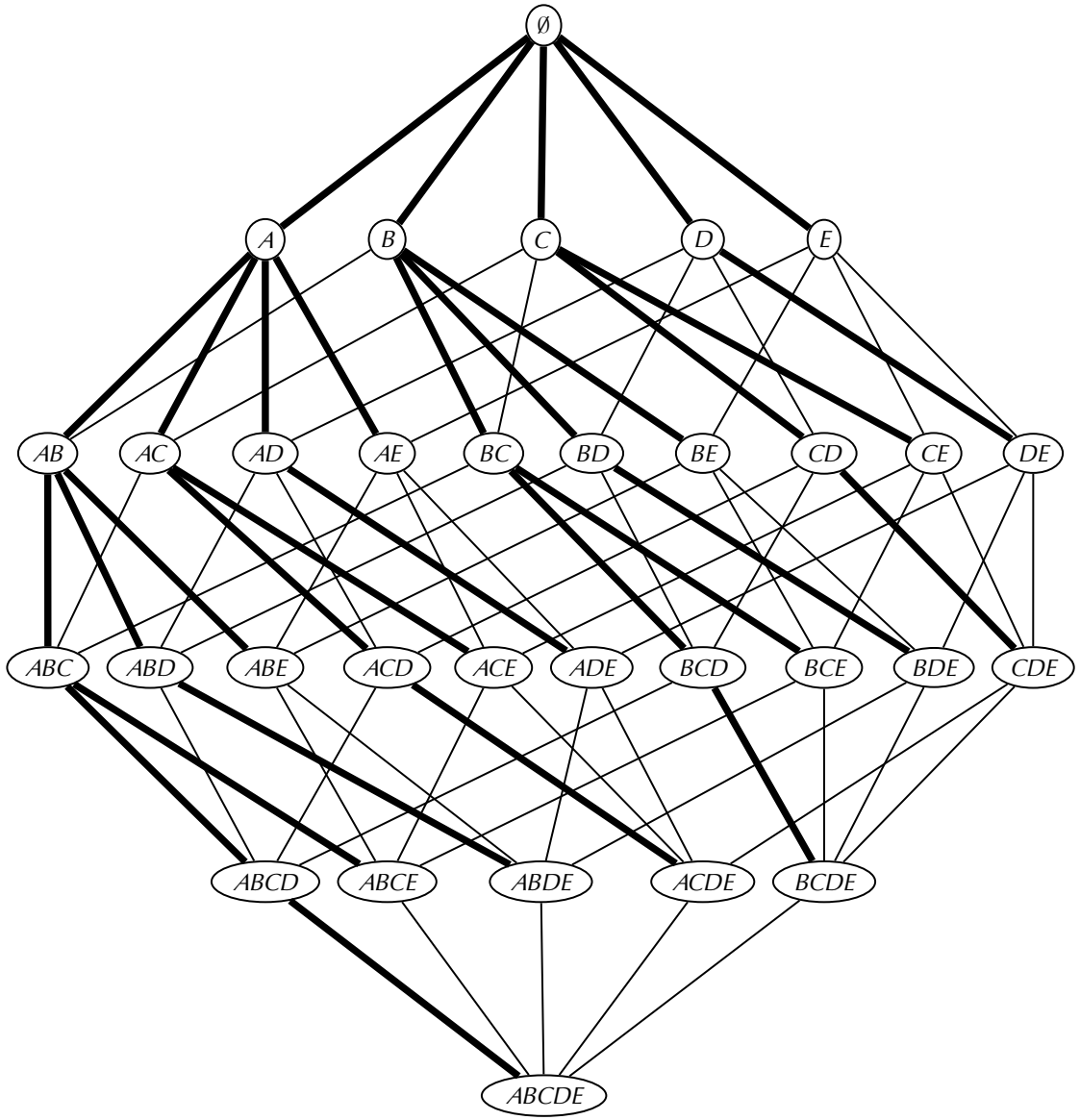
COMPUTESUPPORT (X, D):

```
1 sup(X) ← 0
2 foreach ⟨t, i(t)⟩ ∈ D do
3     if X ⊆ i(t) then
4         sup(X) ← sup(X) + 1
5 return sup(X)
```

Itemset lattice and prefix-based search tree

Itemset search space is a lattice where any two itemsets X and Y are connected by a link iff X is an *immediate subset* of Y, that is, $X \subseteq Y$ and $|X| = |Y| - 1$.

Frequent itemsets can enumerated using either a BFS or DFS search on the *prefix tree*, where two itemsets X, Y are connected by a link iff X is an immediate subset and prefix of Y. This allows one to enumerate itemsets starting with an empty set, and adding one more item at a time.



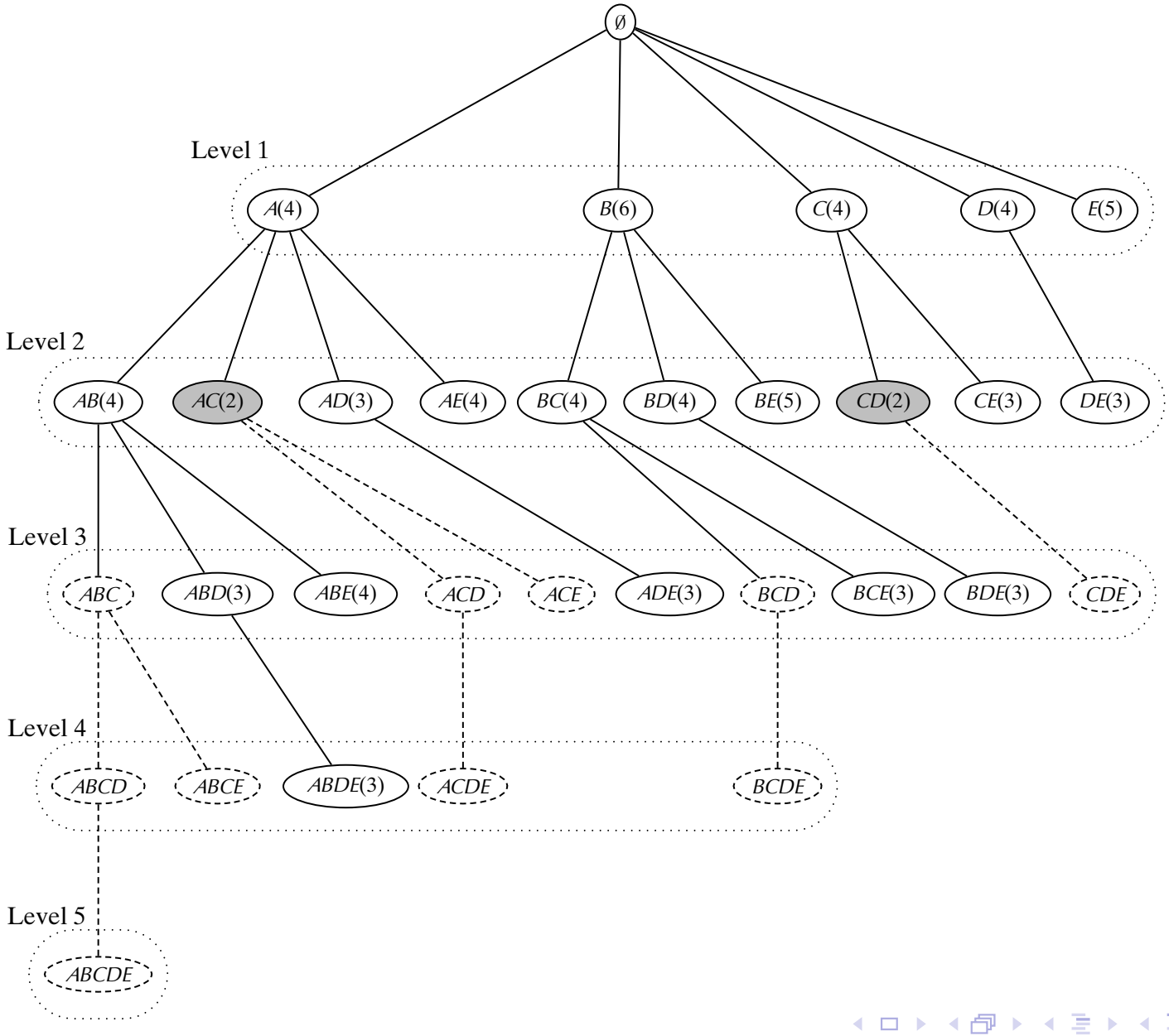
Level-wise Approach: Apriori Algorithm

If $X \subseteq Y$, then $sup(X) \geq sup(Y)$, which leads to the following two observations: (1) if X is frequent, then any subset $Y \subseteq X$ is also frequent, and (2) if X is not frequent, then any superset $Y \supseteq X$ cannot be frequent.

The *Apriori algorithm* utilizes these two properties to significantly improve the brute-force approach. It employs a level-wise or breadth-first exploration of the itemset search space, and prunes all supersets of any infrequent candidate, as no superset of an infrequent itemset can be frequent. It also avoids generating any candidate that has an infrequent subset.

In addition to improving the candidate generation step via itemset pruning, the Apriori method also significantly improves the I/O complexity. Instead of counting the support for a single itemset, it explores the prefix tree in a breadth-first manner, and computes the support of all the valid candidates of size k that comprise level k in the prefix tree.

Apriori Algorithm: Prefix Search Tree and Pruning



The Apriori Algorithm

APRIORI (D, $\mathcal{I}$, *minsup*):

1 $\mathcal{F} \leftarrow \emptyset$
2 $\mathcal{C}^{(1)} \leftarrow \{\emptyset\}$ `// Initial prefix tree with single items`
3 **foreach** $i \in \mathcal{I}$ **do** Add i as child of $\emptyset$ in $\mathcal{C}^{(1)}$ with $sup(i) \leftarrow 0$
4 $k \leftarrow 1$ `//` k `denotes the level`
5 **while** $\mathcal{C}^{(k)} \neq \emptyset$ **do**
6 COMPUTESUPPORT $(\mathcal{C}^{(k)}, \mathbf{D})$
7 **foreach** *leaf* $X \in \mathcal{C}^{(k)}$ **do**
8 **if** $sup(X) \geq minsup$ **then** $\mathcal{F} \leftarrow \mathcal{F} \cup \{(X, sup(X))\}$
9 **else** remove X from $\mathcal{C}^{(k)}$
10 $\mathcal{C}^{(k+1)} \leftarrow$ EXTENDPREFIXTREE $(\mathcal{C}^{(k)})$
11 $k \leftarrow k+1$
12 **return** $\mathcal{F}^{(k)}$

Apriori Algorithm

COMPUTESUPPORT ($\mathcal{C}^{(k)}$, **D**):

```
1 foreach ⟨t, i(t)⟩ ∈ D do
2     foreach k-subset X ⊆ i(t) do
3         if X ∈ C^(k) then sup(X) ← sup(X) + 1
```

EXTENDPREFIXTREE ($\mathcal{C}^{(k)}$):

```
1 foreach leaf X_a ∈ C^(k) do
2     foreach leaf X_b ∈ SIBLING(X_a), such that b > a do
3         X_ab ← X_a ∪ X_b
          // prune candidate if there are any infrequent
             subsets
4         if X_j ∈ C^(k), for all X_j ⊂ X_ab, such that |X_j| = |X_ab| − 1 then
5             Add X_ab as child of X_a with sup(X_ab) ← 0
6     if no extensions from X_a then
7         remove X_a, and all ancestors of X_a with no extensions, from C^(k)
8 return C^(k)
```

Apriori Algorithm: Details

Let $\mathcal{C}^{(k)}$ denote the prefix tree comprising all the candidate k-itemsets.

Apriori begins by inserting the single items into an initially empty prefix tree to populate $\mathcal{C}^{(1)}$.

The support for the current candidates is obtained via COMPUTESUPPORT procedure that generates k-subsets of each transaction in the database $\mathbf{D}$, and for each such subset it increments the support of the corresponding candidate in $\mathcal{C}^{(k)}$ if it exists. Next, we remove any infrequent candidate.

The leaves of the prefix tree that survive comprise the set of frequent k-itemsets $\mathcal{F}^{(k)}$, which are used to generate the candidate $(k+1)$-itemsets for the next level. The EXTENDPREFIXTREE procedure employs prefix-based extension for candidate generation. Given two frequent k-itemsets X_a and X_b with a common $k-1$ length prefix, that is, given two sibling leaf nodes with a common parent, we generate the $(k+1)$-length candidate $X_{ab} = X_a \cup X_b$. This candidate is retained only if it has no infrequent subset. Finally, if a k-itemset X_a has no extension, it is pruned from the prefix tree, and we recursively prune any of its ancestors with no k-itemset extension, so that in $\mathcal{C}^{(k)}$ all leaves are at level k.

If new candidates were added, the whole process is repeated for the next level. This process continues until no new candidates are added.

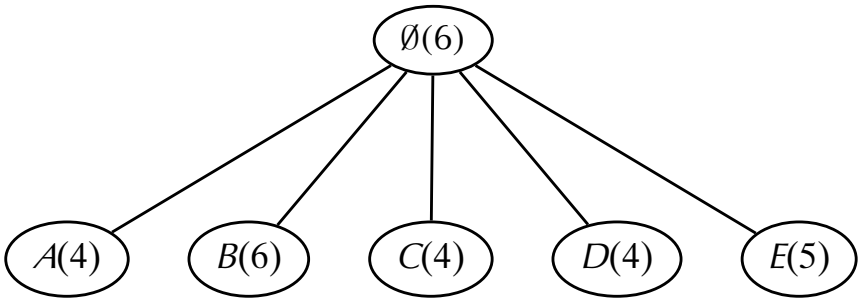
Itemset Mining: Apriori Algorithm

Infrequent itemsets in gray

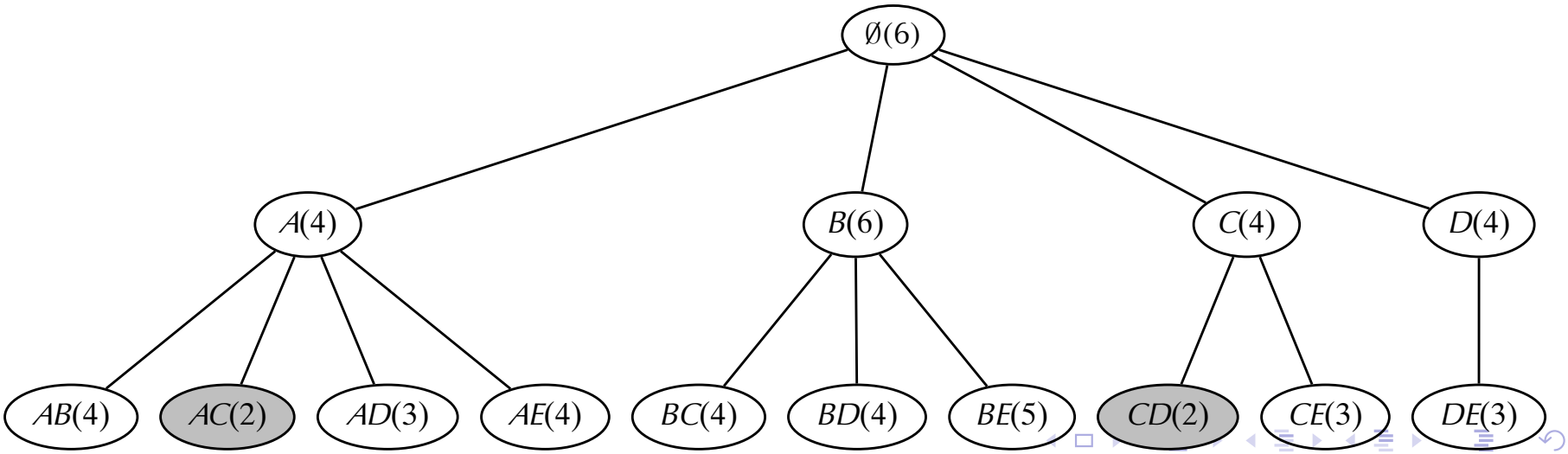
D

t	$\mathbf{i}(t)$
1	$ABDE$
2	BCE
3	$ABDE$
4	$ABCE$
5	$ABCDE$
6	BCD

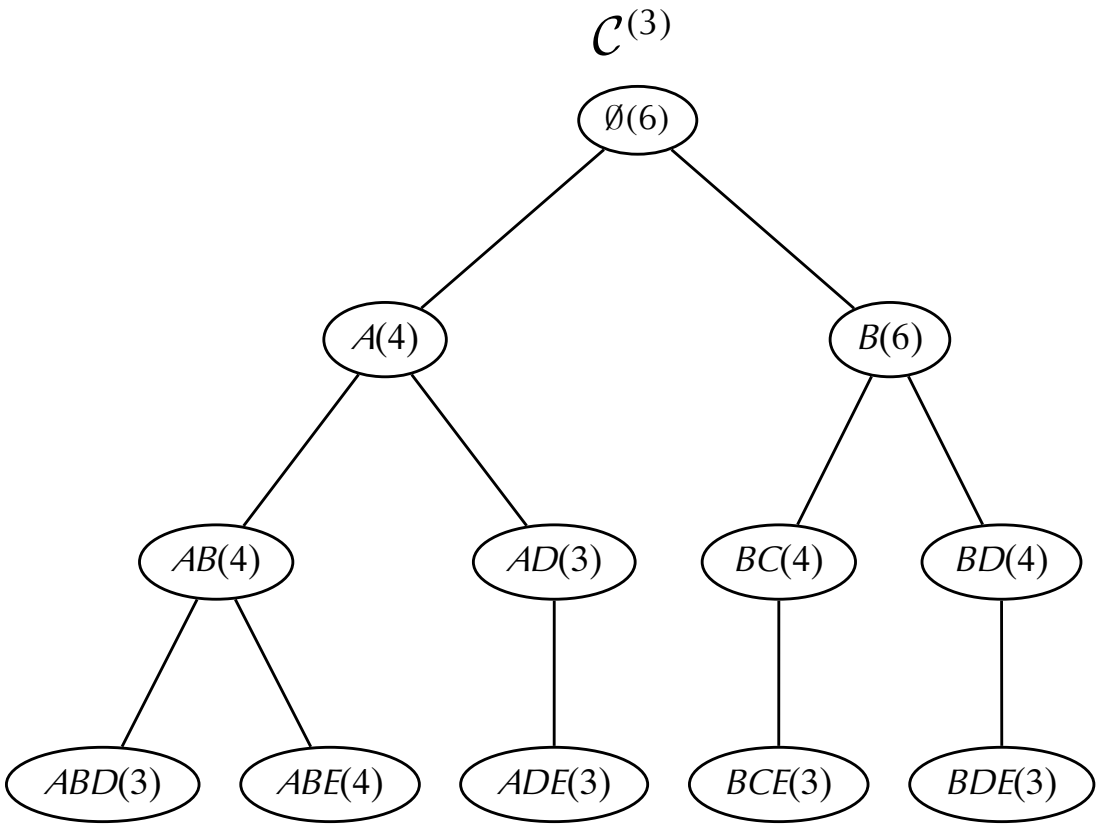
$\mathcal{C}^{(1)}$

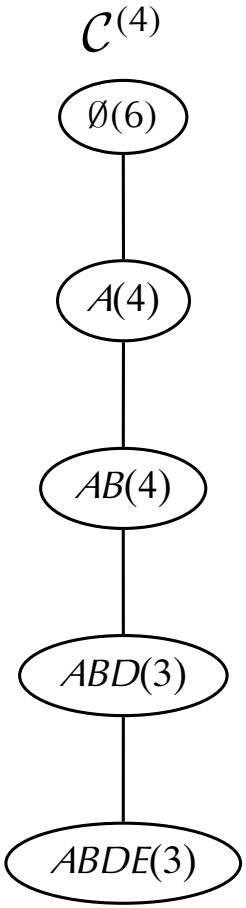


$\mathcal{C}^{(2)}$



Itemset Mining: Apriori Algorithm





Tidset Intersection Approach: Eclat Algorithm

The support counting step can be improved significantly if we can index the database in such a way that it allows fast frequency computations.

The Eclat algorithm leverages the tidsets directly for support computation. The basic idea is that the support of a candidate itemset can be computed by intersecting the tidsets of suitably chosen subsets. In general, given $\mathbf{t}(X)$ and $\mathbf{t}(Y)$ for any two frequent itemsets X and Y, we have

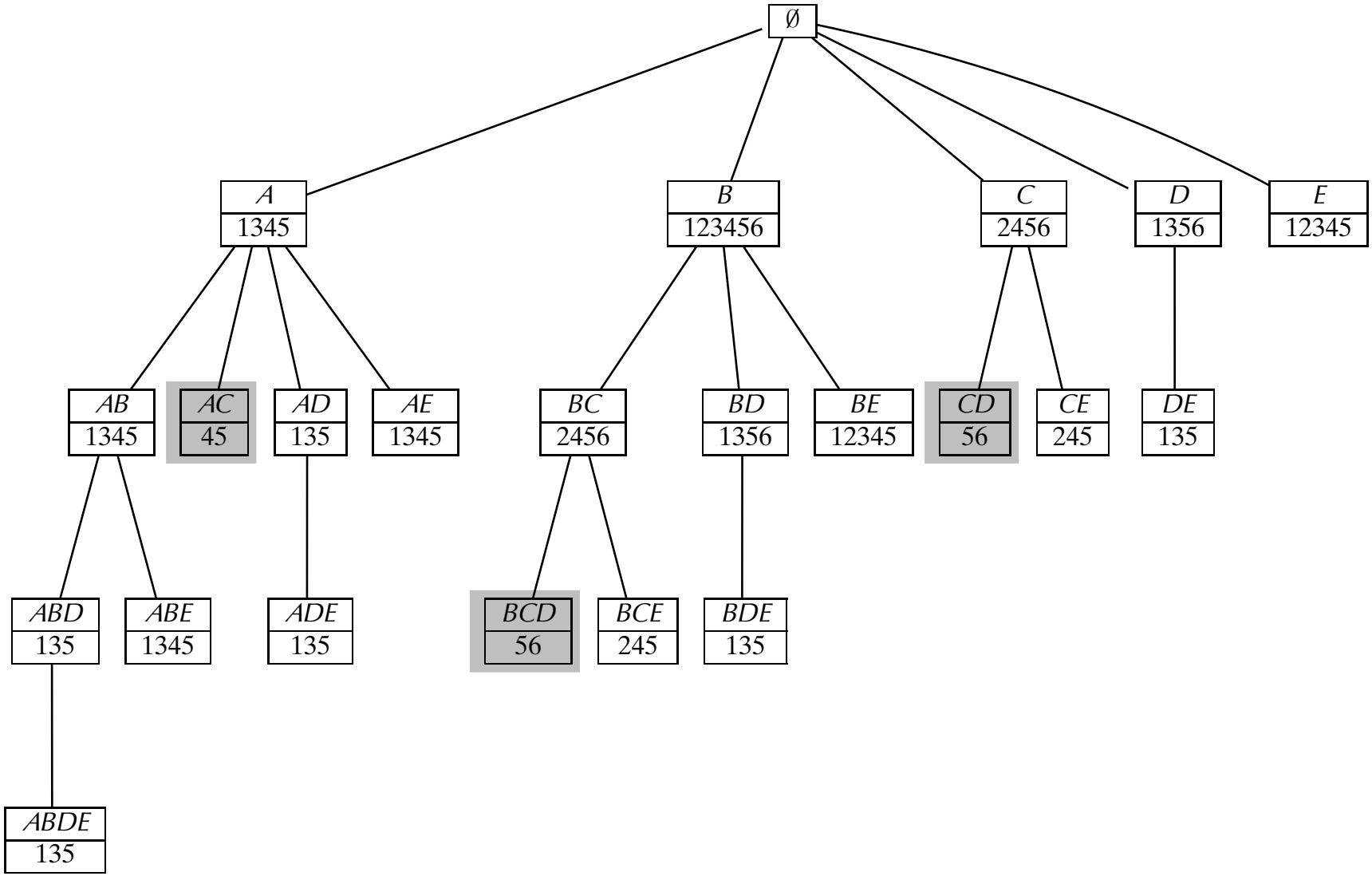
$$\mathbf{t}(XY) = \mathbf{t}(X) \cap \mathbf{t}(Y)$$

The support of candidate XY is simply the cardinality of $\mathbf{t}(XY)$, that is, $sup(XY) = |\mathbf{t}(XY)|$.

Eclat intersects the tidsets only if the frequent itemsets share a common prefix, and it traverses the prefix search tree in a DFS-like manner, processing a group of itemsets that have the same prefix, also called a *prefix equivalence class*.

Eclat Algorithm: Tidlist Intersections

Infrequent itemsets in gray



Eclat Algorithm

// Initial Call: $\mathcal{F} \leftarrow \emptyset, P \leftarrow \{\langle i, \mathbf{t}(i)\rangle \mid i \in \mathcal{I}, |\mathbf{t}(i)| \geq minsup\}$

ECLAT (P, $minsup$, $\mathcal{F}$):

```
1 foreach ⟨X_a, t(X_a)⟩ ∈ P do
2     F ← F ∪ {(X_a, sup(X_a))}
3     P_a ← ∅
4     foreach ⟨X_b, t(X_b)⟩ ∈ P, with X_b > X_a do
5         X_ab = X_a ∪ X_b
6         t(X_ab) = t(X_a) ∩ t(X_b)
7         if sup(X_ab) ≥ minsup then
8             P_a ← P_a ∪ {⟨X_ab, t(X_ab)⟩}
9     if P_a ≠ ∅ then ECLAT (P_a, minsup, F)
```

Diffsets: Difference of Tidsets

The Eclat algorithm can be significantly improved if we can shrink the size of the intermediate tidsets. This can be achieved by keeping track of the differences in the tidsets as opposed to the full tidsets.

Let $X_a = \{x_1, \ldots, x_{k-1}, x_a\}$ and $X_b = \{x_1, \ldots, x_{k-1}, x_b\}$, so that $X_{ab} = X_a \cup X_b = \{x_1, \ldots, x_{k-1}, x_a, x_b\}$.

The *diffset* of X_{ab} is the set of tids that contain the prefix X_a, but not the item X_b

$$\mathbf{d}(X_{ab}) = \mathbf{t}(X_a) \setminus \mathbf{t}(X_{ab}) = \mathbf{t}(X_a) \setminus \mathbf{t}(X_b)$$

We can obtain an expression for $\mathbf{d}(X_{ab})$ in terms of $\mathbf{d}(X_a)$ and $\mathbf{d}(X_b)$ as follows:

$$\mathbf{d}(X_{ab}) = \mathbf{d}(X_b) \setminus \mathbf{d}(X_a)$$

which means that we can replace all intersection operations in Eclat with diffset operations.

Algorithm dEclat

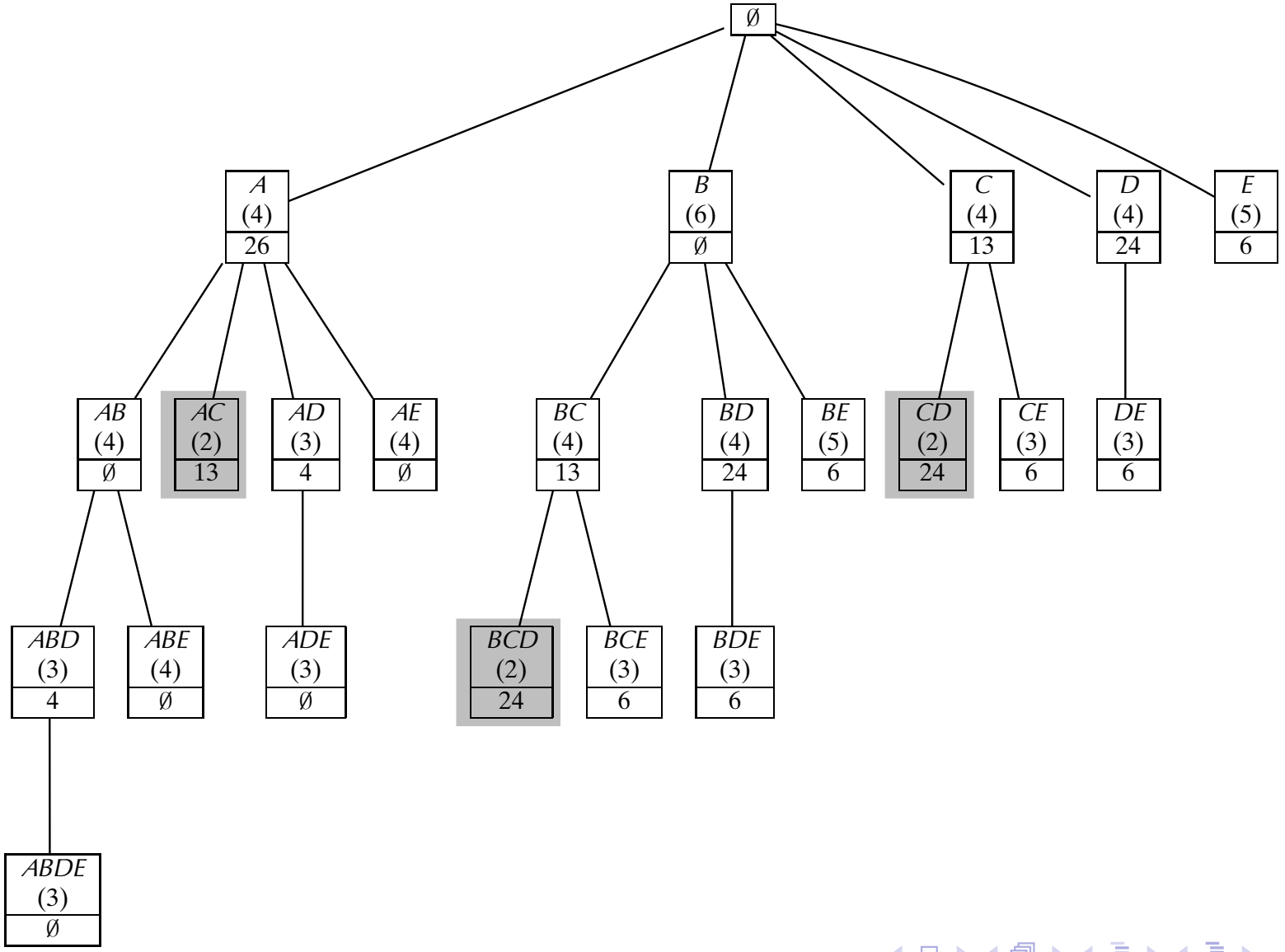
// Initial Call: $\mathcal{F} \leftarrow \emptyset$,
$P \leftarrow \{\langle i, \mathbf{d}(i), sup(i)\rangle \mid i \in \mathcal{I}, \mathbf{d}(i) = \mathcal{T} \setminus \mathbf{t}(i), sup(i) \geq minsup\}$

dEclat (P, $minsup$, $\mathcal{F}$):

foreach $\langle X_a, \mathbf{d}(X_a), sup(X_a)\rangle \in P$ **do**
  $\mathcal{F} \leftarrow \mathcal{F} \cup \{(X_a, sup(X_a))\}$
  $P_a \leftarrow \emptyset$
  foreach $\langle X_b, \mathbf{d}(X_b), sup(X_b)\rangle \in P$, *with* $X_b > X_a$ **do**
    $X_{ab} = X_a \cup X_b$
    $\mathbf{d}(X_{ab}) = \mathbf{d}(X_b) \setminus \mathbf{d}(X_a)$
    $sup(X_{ab}) = sup(X_a) - |\mathbf{d}(X_{ab})|$
    if $sup(X_{ab}) \geq minsup$ **then**
      $P_a \leftarrow P_a \cup \{\langle X_{ab}, \mathbf{d}(X_{ab}), sup(X_{ab})\rangle\}$
  if $P_a \neq \emptyset$ **then** dEclat $(P_a, minsup, \mathcal{F})$

dEclat Algorithm: Diffsets

support shown within brackets; infrequent itemsets in gray



Frequent Pattern Tree Approach: FPGrowth Algorithm

The FPGrowth method indexes the database for fast support computation via the use of an augmented prefix tree called the *frequent pattern tree* (FP-tree).

Each node in the tree is labeled with a single item, and each child node represents a different item. Each node also stores the support information for the itemset comprising the items on the path from the root to that node.

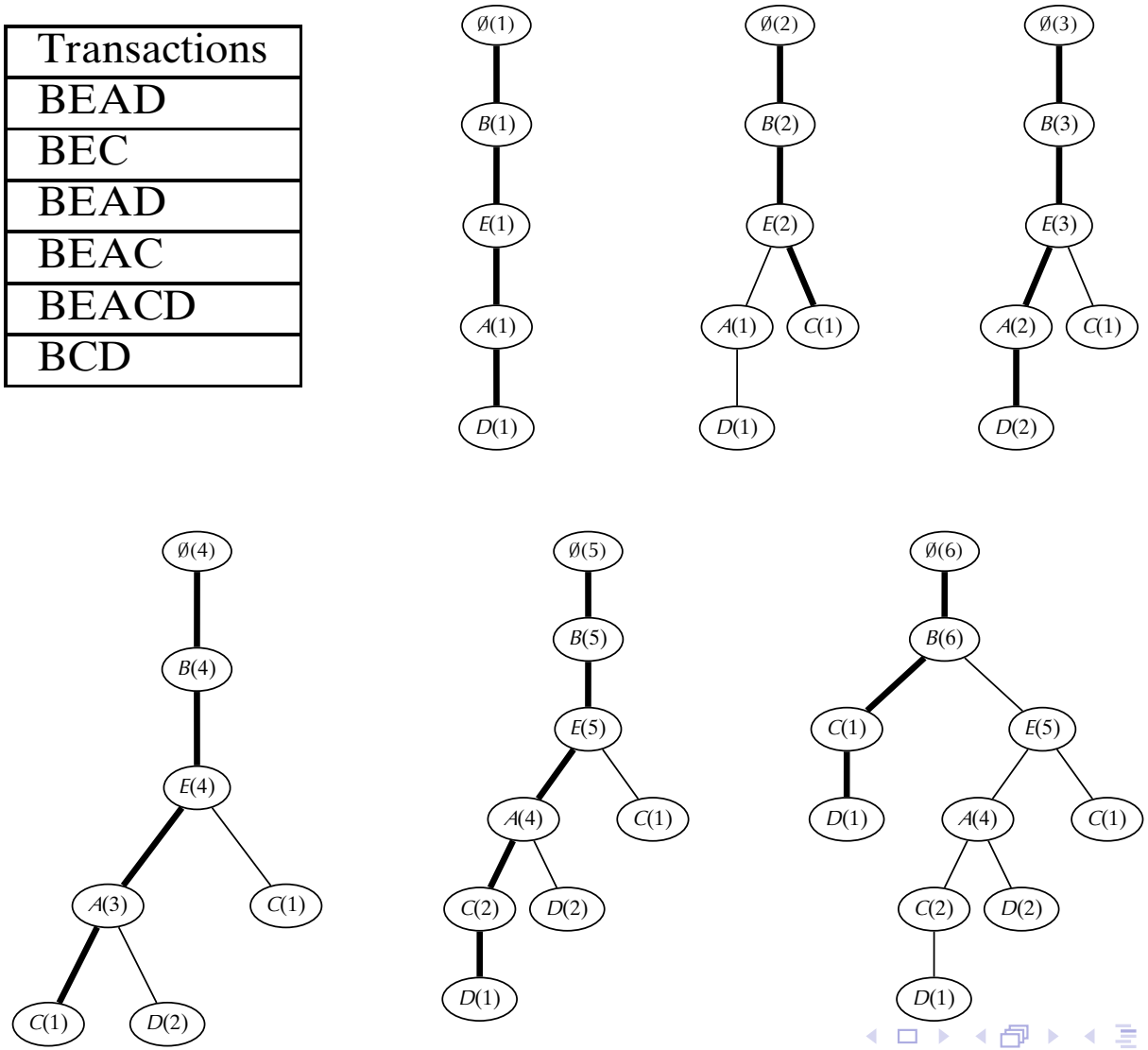
The FP-tree is constructed as follows. Initially the tree contains as root the null item $\emptyset$. Next, for each tuple $\langle t, X \rangle \in \mathbf{D}$, where $X = \mathbf{i}(t)$, we insert the itemset X into the FP-tree, incrementing the count of all nodes along the path that represents X.

If X shares a prefix with some previously inserted transaction, then X will follow the same path until the common prefix. For the remaining items in X, new nodes are created under the common prefix, with counts initialized to 1. The FP-tree is complete when all transactions have been inserted.

Frequent Pattern Tree

The FP-tree is a prefix compressed representation of **D**. For most compression items are sorted in descending order of support.

Transactions
BEAD
BEC
BEAD
BEAC
BEACD
BCD



FPGrowth Algorithm: Details

Given an FP-tree R, projected FP-trees are built for each frequent item i in R in increasing order of support in a recursive manner.

To project R on item i, we find all the occurrences of i in the tree, and for each occurrence, we determine the corresponding path from the root to i. The count of item i on a given path is recorded in $cnt(i)$ and the path is inserted into the new projected tree R_X, where X is the itemset obtained by extending the prefix P with the item i. While inserting the path, the count of each node in R_X along the given path is incremented by the path count $cnt(i)$.

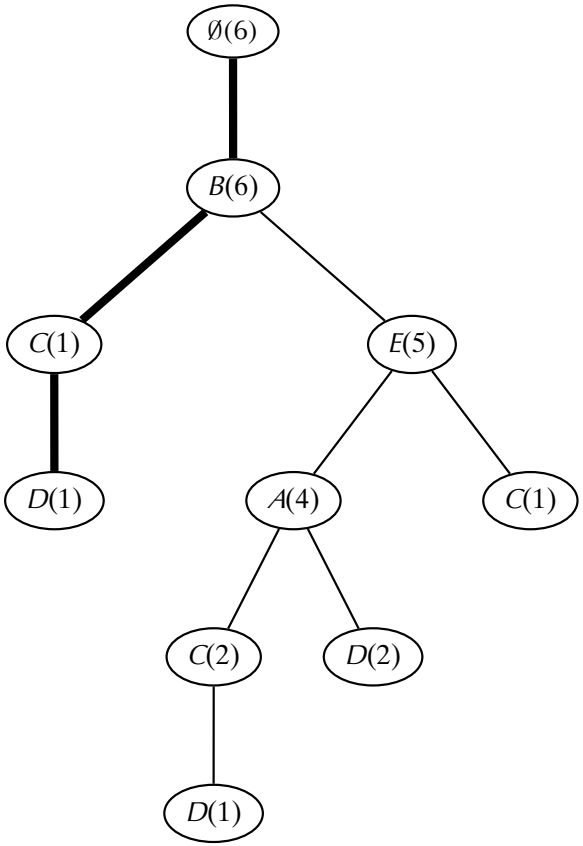
The base case for the recursion happens when the input FP-tree R is a single path. FP-trees that are paths are handled by enumerating all itemsets that are subsets of the path, with the support of each such itemset being given by the least frequent item in it.

FPGrowth Algorithm

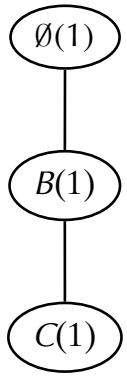
```
// Initial Call:  R ← FP-tree(D), P ← ∅, F ← ∅
FPGROWTH (R, P, F, minsup):
1  Remove infrequent items from R
2  if ISPATH(R) then // insert subsets of R into F
3      foreach Y ⊆ R do
4          X ← P ∪ Y
5          sup(X) ← min_{x∈Y}{cnt(x)}
6          F ← F ∪ {(X, sup(X))}
7  else  // process projected FP-trees for each frequent item i
8      foreach i ∈ R in increasing order of sup(i) do
9          X ← P ∪ {i}
10         sup(X) ← sup(i) // sum of cnt(i) for all nodes labeled i
11         F ← F ∪ {(X, sup(X))}
12         R_X ← ∅ // projected FP-tree for X
13         foreach path ∈ PATHFROMROOT(i) do
14             cnt(i) ← count of i in path
15             Insert path, excluding i, into FP-tree R_X with count cnt(i)
16         if R_X ≠ ∅ then FPGROWTH (R_X, X, F, minsup)
```

Projected Frequent Pattern Tree for D

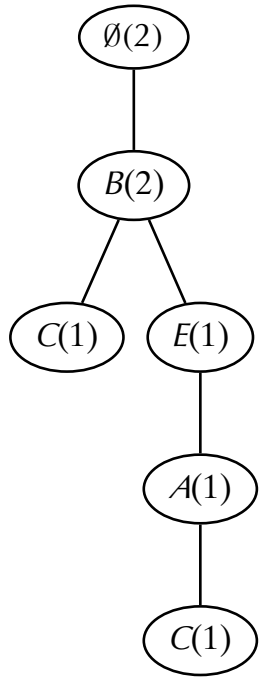
FP-Tree



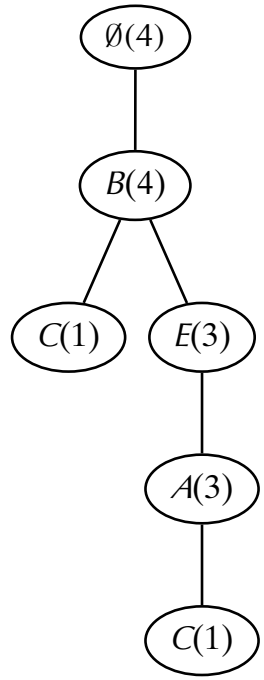
Add BC, $cnt = 1$



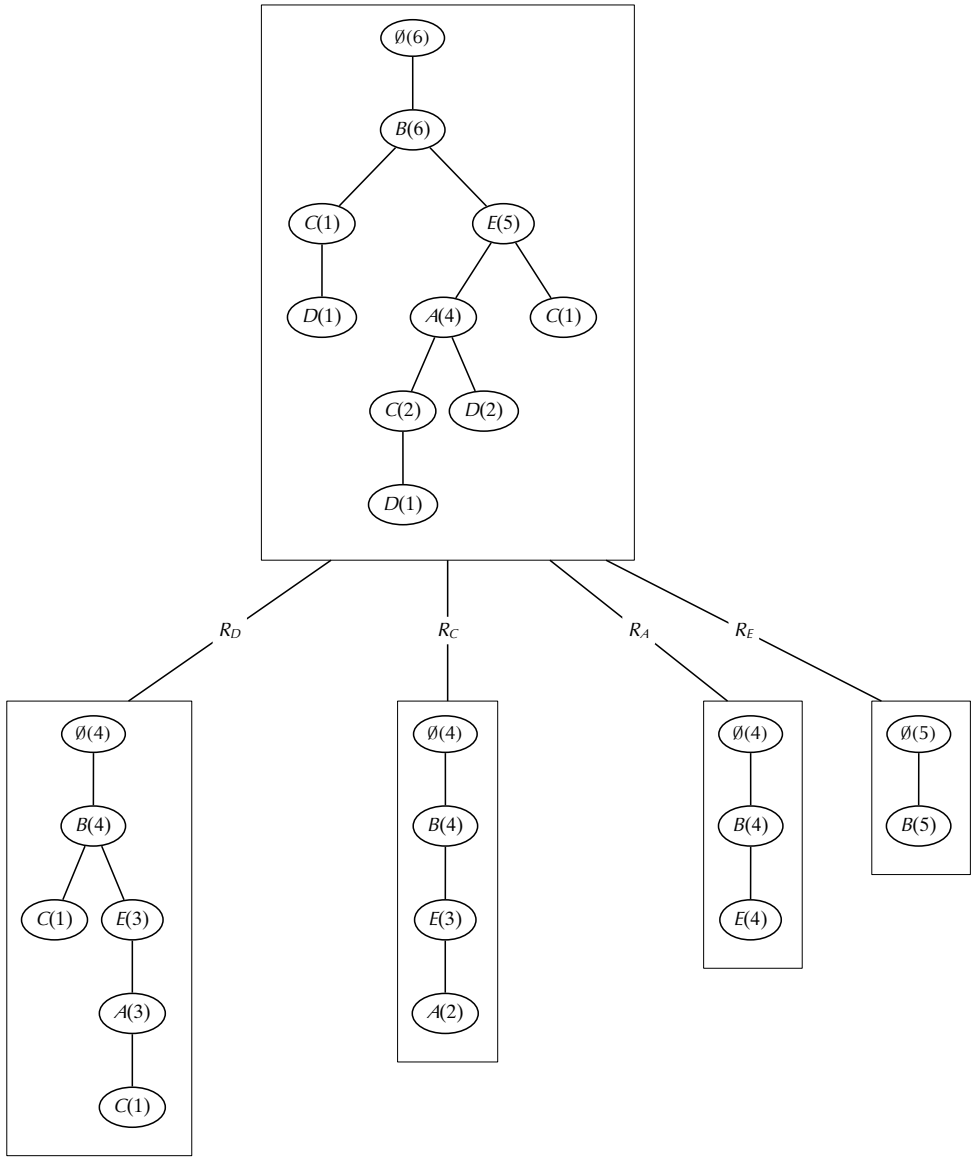
Add $BEAC$, $cnt = 1$



Add BEA, $cnt = 2$



FPGrowth: Frequent Pattern Tree Projection



Generating Association Rules

Given a frequent itemset $Z \in \mathcal{F}$, we look at all proper subsets $X \subset Z$ to compute rules of the form

$$X \xrightarrow{s,c} Y, \text{ where } Y = Z \setminus X$$

where $Z \setminus X = Z - X$.

The rule must be frequent because

$$s = sup(XY) = sup(Z) \geq minsup$$

We compute the confidence as follows:

$$c = \frac{sup(X \cup Y)}{sup(X)} = \frac{sup(Z)}{sup(X)}$$

If $c \geq minconf$, then the rule is a strong rule. On the other hand, if $conf(X \longrightarrow Y) < c$, then $conf(W \longrightarrow Z \setminus W) < c$ for all subsets $W \subset X$, as $sup(W) \geq sup(X)$. We can thus avoid checking subsets of X.

Association Rule Mining Algorithm

AssociationRules ($\mathcal{F}$, *minconf*):

1 **foreach** $Z \in \mathcal{F}$, *such that* $|Z| \geq 2$ **do**
2 $\mathcal{A} \leftarrow \{X \mid X \subset Z, X \neq \emptyset\}$
3 **while** $\mathcal{A} \neq \emptyset$ **do**
4 $X \leftarrow$ maximal element in $\mathcal{A}$
5 $\mathcal{A} \leftarrow \mathcal{A} \setminus X$ // `remove` X `from` $\mathcal{A}$
6 $c \leftarrow sup(Z)/sup(X)$
7 **if** $c \geq minconf$ **then**
8 print $X \longrightarrow Y, sup(Z), c$
9 **else**
10 $\mathcal{A} \leftarrow \mathcal{A} \setminus \{W \mid W \subset X\}$
 // `remove all subsets of` X `from` $\mathcal{A}$

- Frequent Itemset Mining
- **k-Means**
- DBScan
- Decision trees

Representative-based Clustering

Given a dataset with n points in a d-dimensional space, $\mathbf{D} = \{\mathbf{x}_i\}_{i=1}^n$, and given the number of desired clusters k, the goal of representative-based clustering is to partition the dataset into k groups or clusters, which is called a *clustering* and is denoted as $\mathcal{C} = \{C_1, C_2, \ldots, C_k\}$.

For each cluster C_i there exists a representative point that summarizes the cluster, a common choice being the mean (also called the *centroid*) $\boldsymbol{\mu}_i$ of all points in the cluster, that is,

$$\boldsymbol{\mu}_i = \frac{1}{n_i} \sum_{x_j \in C_i} \mathbf{x}_j$$

where $n_i = |C_i|$ is the number of points in cluster C_i.

A brute-force or exhaustive algorithm for finding a good clustering is simply to generate all possible partitions of n points into k clusters, evaluate some optimization score for each of them, and retain the clustering that yields the best score. However, this is clearly infeasilbe, since there are $O(k^n/k!)$ clusterings of n points into k groups.

K-means Algorithm: Objective

The *sum of squared errors* scoring function is defined as

$$SSE(\mathcal{C}) = \sum_{i=1}^{k} \sum_{\mathbf{x}_j \in C_i} \left\| \mathbf{x}_j - \boldsymbol{\mu}_i \right\|^2$$

The goal is to find the clustering that minimizes the SSE score:

$$\mathcal{C}^* = \arg\min_{\mathcal{C}} \{SSE(\mathcal{C})\}$$

K-means employs a greedy iterative approach to find a clustering that minimizes the SSE objective. As such it can converge to a local optima instead of a globally optimal clustering.

K-means Algorithm: Objective

K-means initializes the cluster means by randomly generating k points in the data space. Each iteration of K-means consists of two steps: (1) cluster assignment, and (2) centroid update.

Given the k cluster means, in the cluster assignment step, each point $\mathbf{x}_j \in \mathbf{D}$ is assigned to the closest mean, which induces a clustering, with each cluster C_i comprising points that are closer to $\boldsymbol{\mu}_i$ than any other cluster mean. That is, each point $\mathbf{x}_j$ is assigned to cluster C_{j^*}, where

$$j^* = \arg\min_{i=1}^{k} \left\{ \left\| \mathbf{x}_j - \boldsymbol{\mu}_i \right\|^2 \right\}$$

Given a set of clusters C_i, $i = 1, \ldots, k$, in the centroid update step, new mean values are computed for each cluster from the points in C_i.

The cluster assignment and centroid update steps are carried out iteratively until we reach a fixed point or local minima.

K-Means Algorithm

K-MEANS $(\mathbf{D}, k, \epsilon)$:

$t = 0$

Randomly initialize k centroids: $\boldsymbol{\mu}_1^t, \boldsymbol{\mu}_2^t, \ldots, \boldsymbol{\mu}_k^t \in \mathbb{R}^d$

repeat

  $t \leftarrow t + 1$

  $C_j \leftarrow \emptyset$ for all $j = 1, \cdots, k$

  `// Cluster Assignment Step`

  foreach $\mathbf{x}_j \in \mathbf{D}$ **do**

    $j^* \leftarrow \arg\min_i \left\{ \left\| \mathbf{x}_j - \boldsymbol{\mu}_i^t \right\|^2 \right\}$ `// Assign` $\mathbf{x}_j$ `to closest centroid`

    $C_{j^*} \leftarrow C_{j^*} \cup \{\mathbf{x}_j\}$

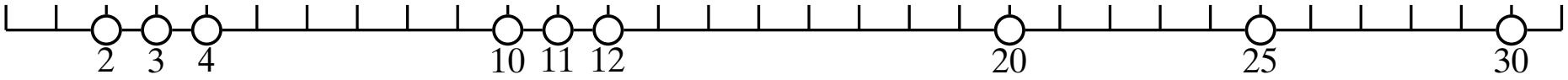
  `// Centroid Update Step`

  foreach $i = 1$ *to* k **do**

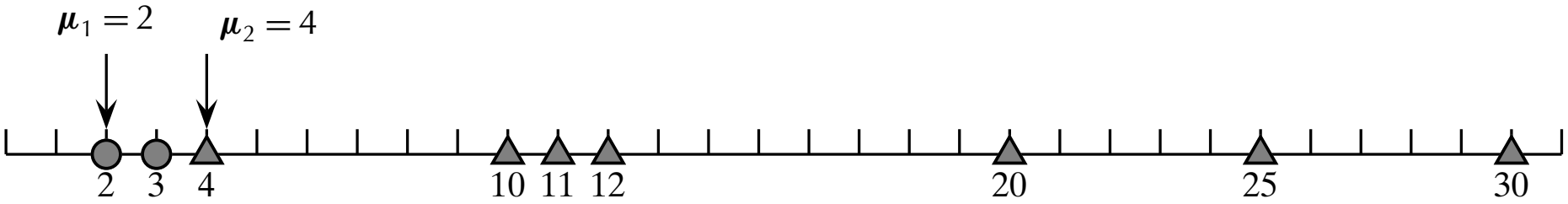
    $\boldsymbol{\mu}_i^t \leftarrow \frac{1}{|C_i|} \sum_{\mathbf{x}_j \in C_i} \mathbf{x}_j$

until $\sum_{i=1}^{k} \left\| \boldsymbol{\mu}_i^t - \boldsymbol{\mu}_i^{t-1} \right\|^2 \leq \epsilon$

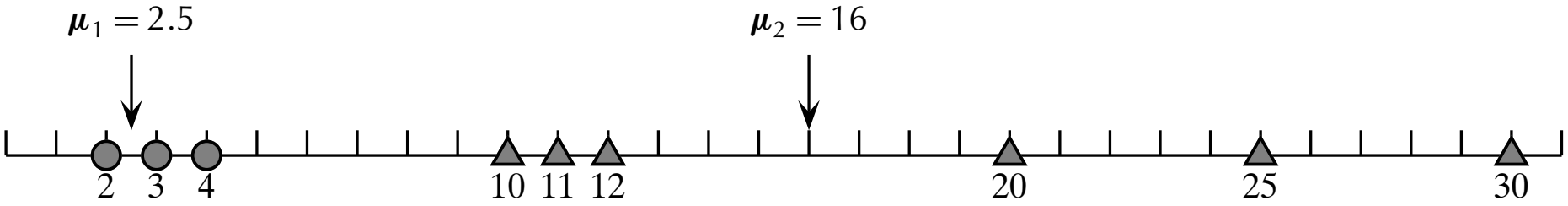
K-means in One Dimension



(a) Initial dataset

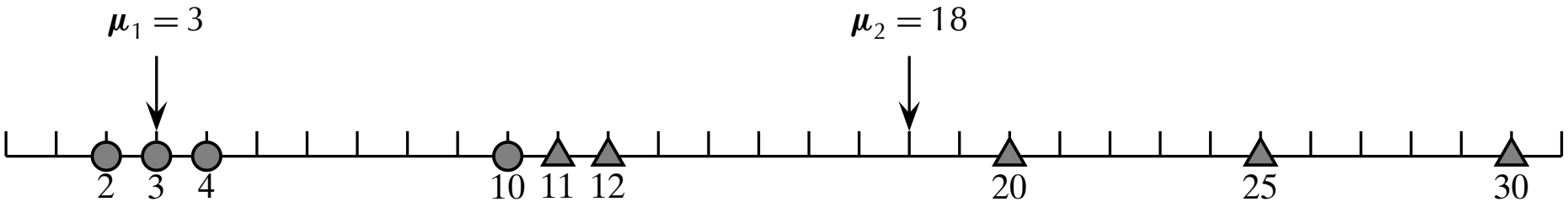


(b) Iteration: $t = 1$

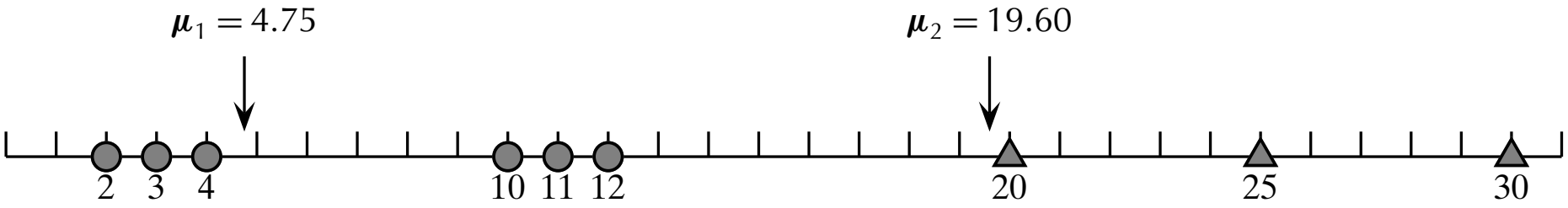


(c) Iteration: $t = 2$

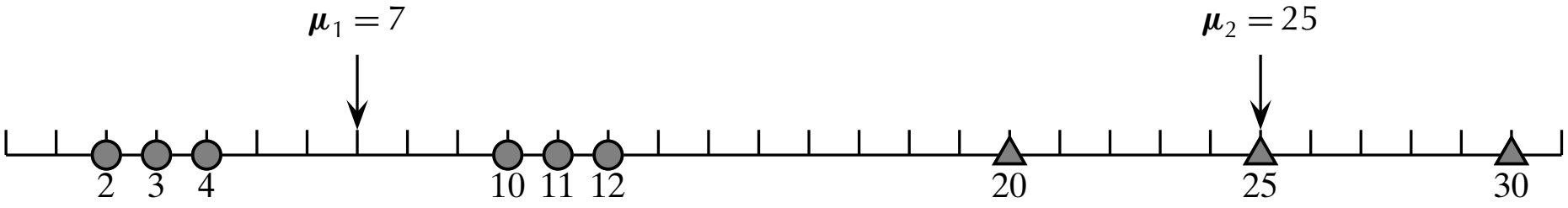
K-means in One Dimension (contd.)



(d) Iteration: $t = 3$

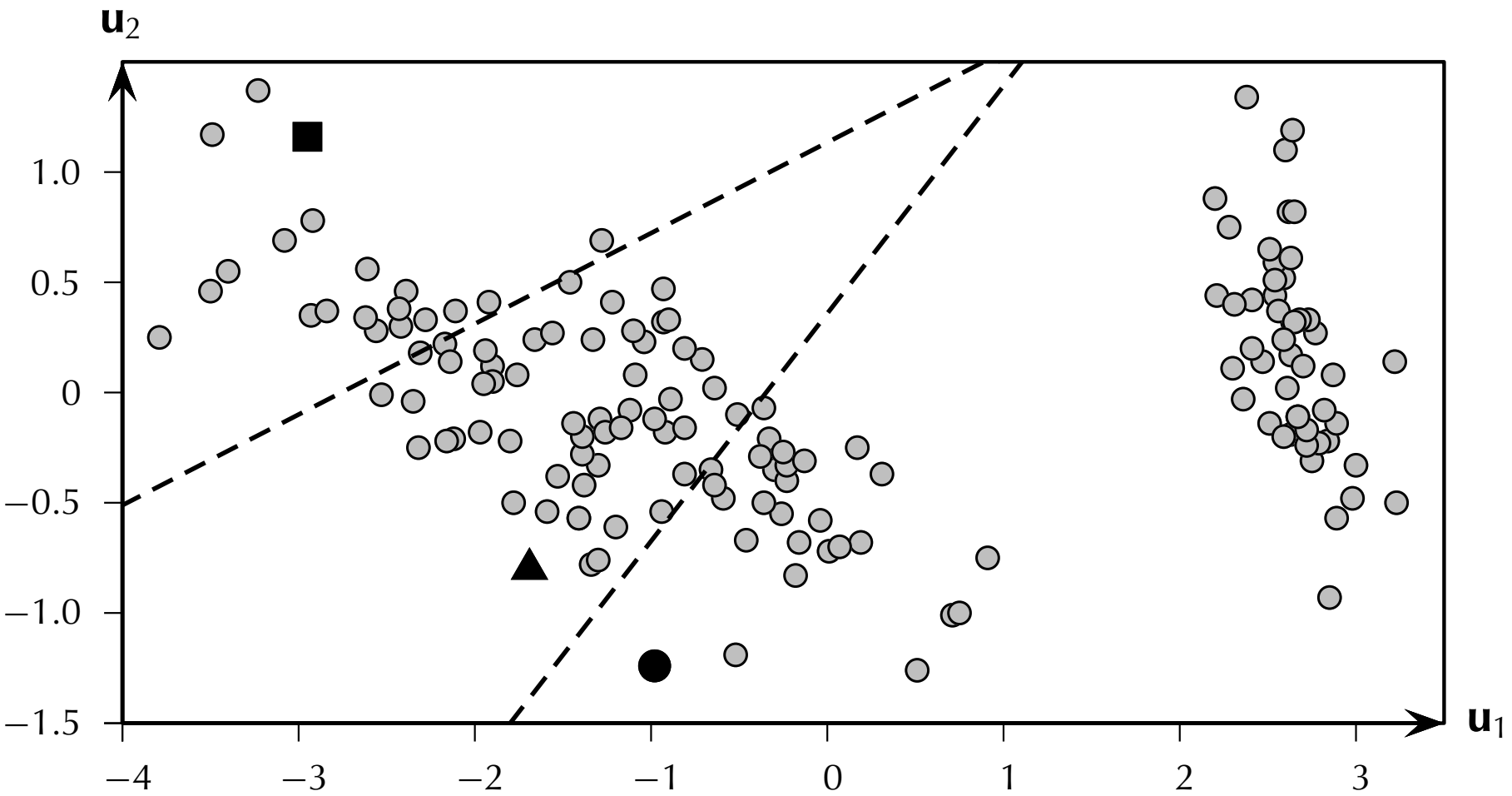


(e) Iteration: $t = 4$



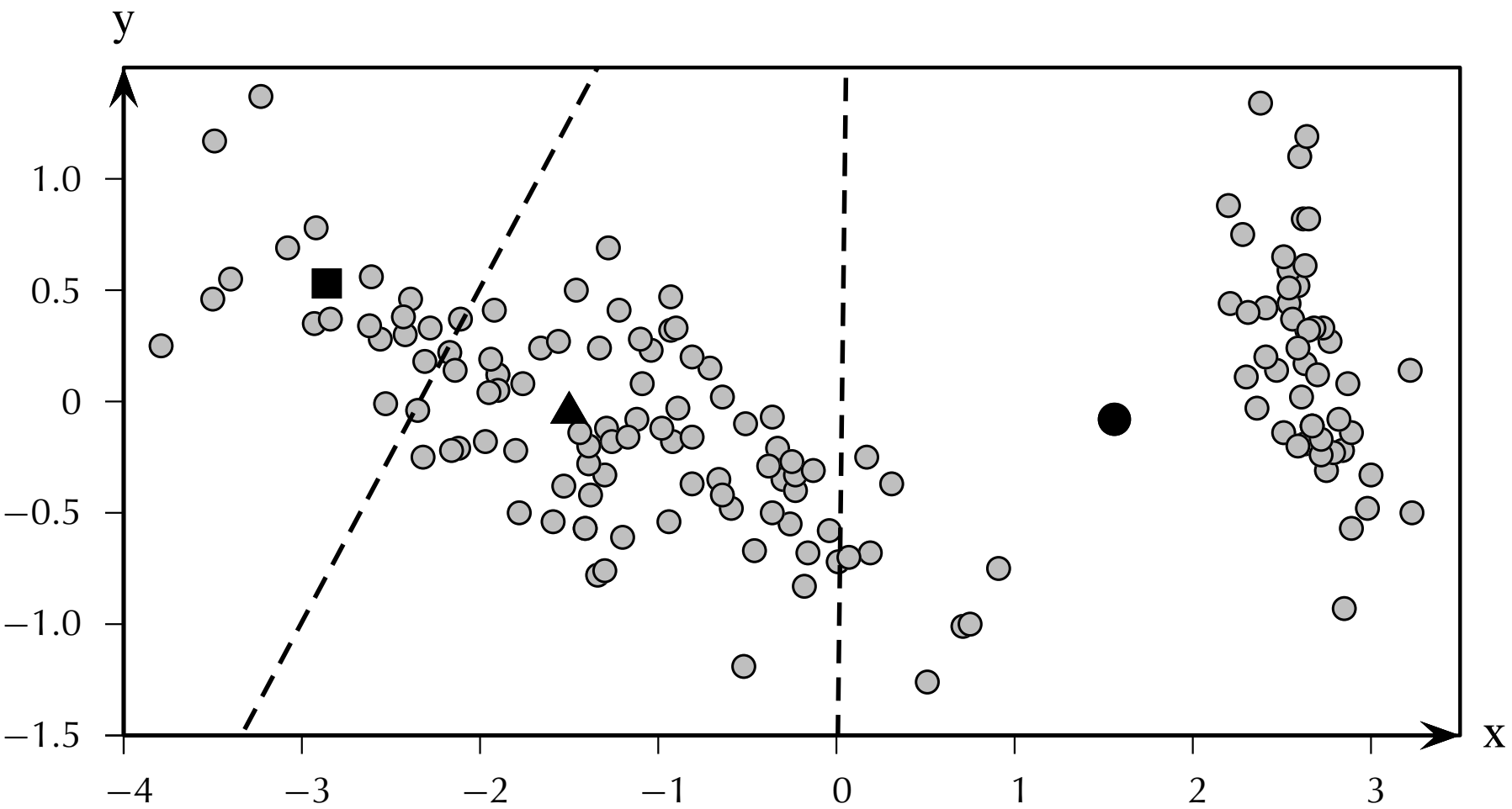
(f) Iteration: $t = 5$ (converged)

K-means in 2D: Iris Principal Components



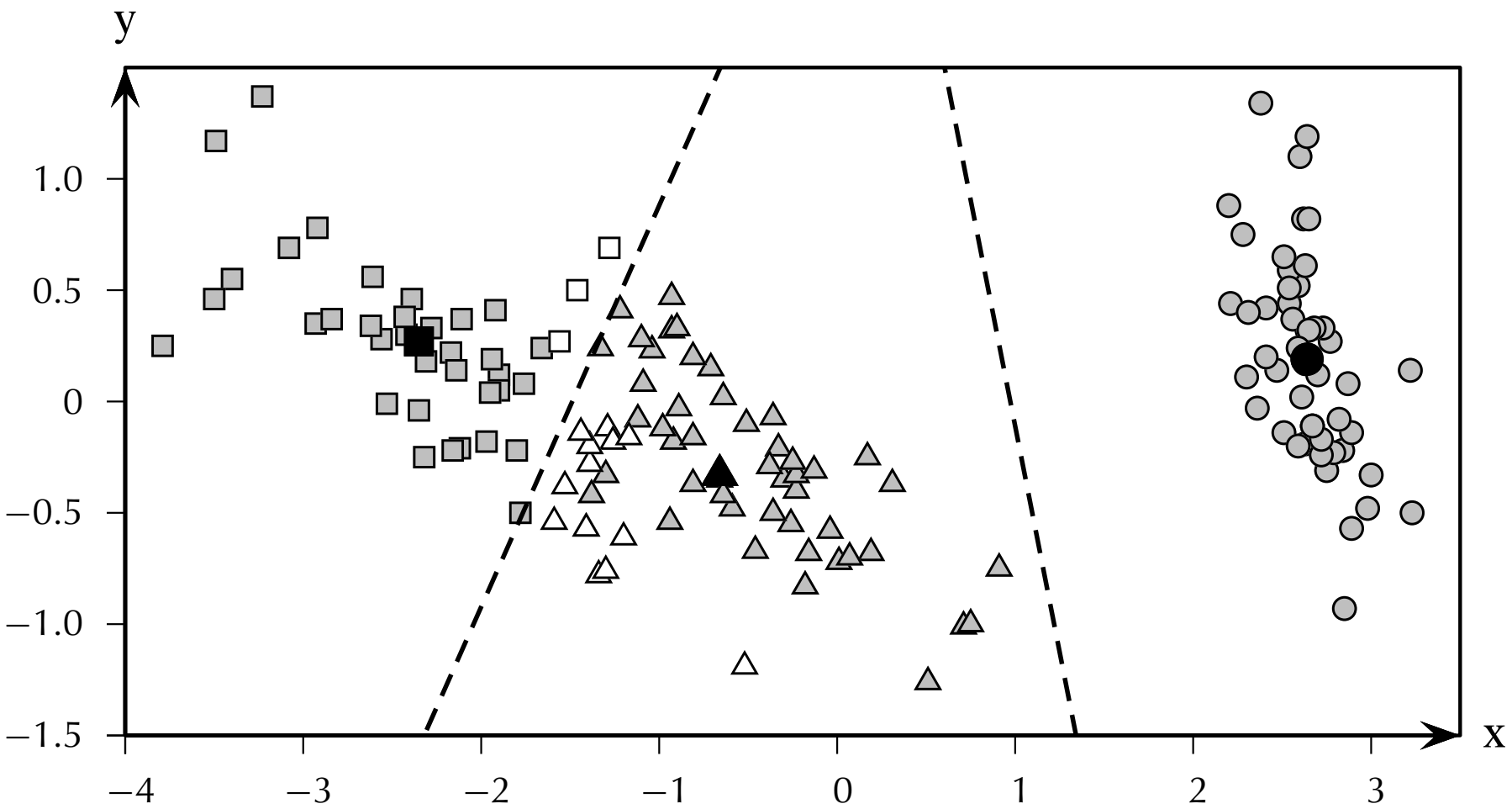
(a) Random initialization: $t = 0$

K-means in 2D: Iris Principal Components



(b) Iteration: $t = 1$

K-means in 2D: Iris Principal Components

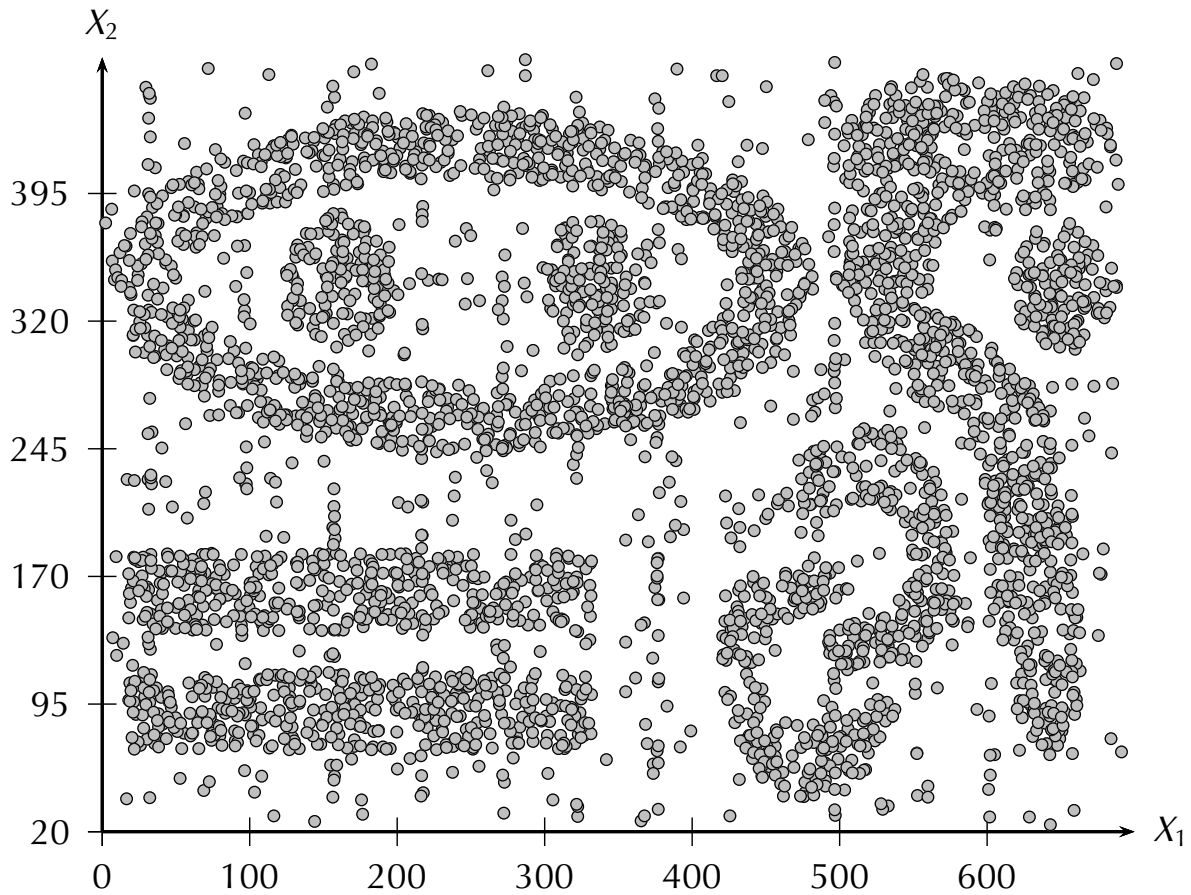


(c) Iteration: $t = 8$ (converged)

- Frequent Itemset Mining
- k-Means
- **DBScan**
- Decision trees

Density-based Clustering

Density-based methods are able to mine nonconvex clusters, where distance-based methods may have difficulty.



The DBSCAN Approach: Neighborhood and Core Points

Define a ball of radius ϵ around a point $\mathbf{x} \in \mathbb{R}^d$, called the ϵ-*neighborhood* of $\mathbf{x}$, as follows:

$$N_\epsilon(\mathbf{x}) = B_d(\mathbf{x}, \epsilon) = \{\mathbf{y} \mid \delta(\mathbf{x}, \mathbf{y}) \leq \epsilon\}$$

Here $\delta(\mathbf{x}, \mathbf{y})$ represents the distance between points $\mathbf{x}$ and $\mathbf{y}$. which is usually assumed to be the Euclidean

We say that $\mathbf{x}$ is a *core point* if there are at least *minpts* points in its ϵ-neighborhood, i.e., if $|N_\epsilon(\mathbf{x})| \geq minpts$.

A *border point* does not meet the *minpts* threshold, i.e., $|N_\epsilon(\mathbf{x})| < minpts$, but it belongs to the ϵ-neighborhood of some core point $\mathbf{z}$, that is, $\mathbf{x} \in N_\epsilon(\mathbf{z})$.

If a point is neither a core nor a border point, then it is called a *noise point* or an outlier.

The DBSCAN Approach: Reachability and Density-based Cluster

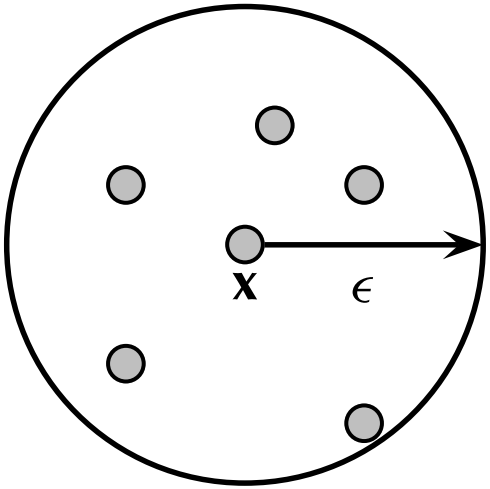
A point $\mathbf{x}$ is *directly density reachable* from another point $\mathbf{y}$ if $\mathbf{x} \in N_\epsilon(\mathbf{y})$ and $\mathbf{y}$ is a core point.

A point $\mathbf{x}$ is *density reachable* from $\mathbf{y}$ if there exists a chain of points, $\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_l$, such that $\mathbf{x} = \mathbf{x}_0$ and $\mathbf{y} = \mathbf{x}_l$, and $\mathbf{x}_i$ is directly density reachable from $\mathbf{x}_{i-1}$ for all $i = 1, \ldots, l$. In other words, there is set of core points leading from $\mathbf{y}$ to $\mathbf{x}$.

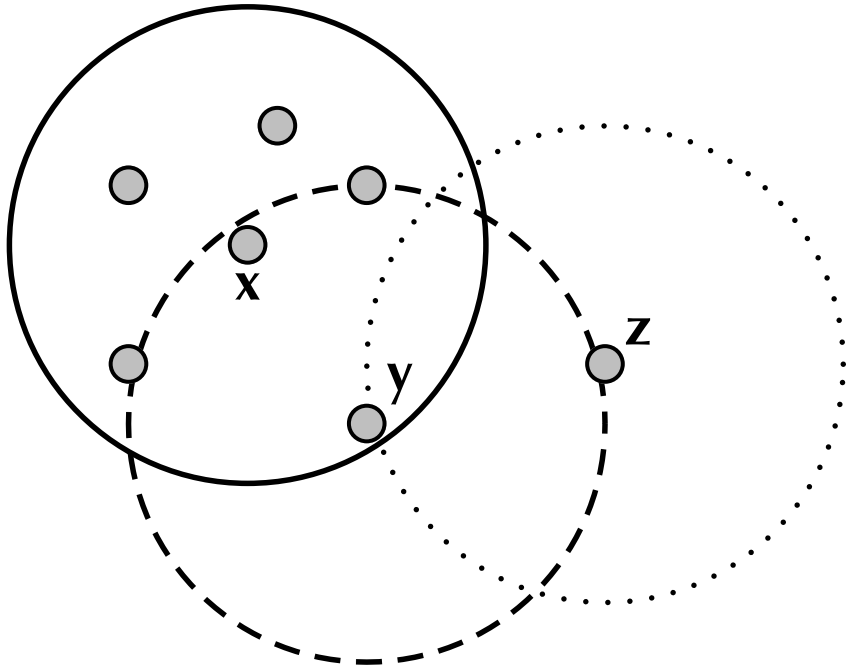
Two points $\mathbf{x}$ and $\mathbf{y}$ are *density connected* if there exists a core point $\mathbf{z}$, such that both $\mathbf{x}$ and $\mathbf{y}$ are density reachable from $\mathbf{z}$.

A *density-based cluster* is defined as a maximal set of density connected points.

Core, Border and Noise Points



(d) Neighborhood of a Point



(e) Core, Border, and Noise Points

DBSCAN Density-based Clustering Algorithm

DBSCAN computes the ϵ-neighborhood $N_\epsilon(\mathbf{x}_i)$ for each point $\mathbf{x}_i$ in the dataset $\mathbf{D}$, and checks if it is a core point. It also sets the cluster id $id(\mathbf{x}_i) = \emptyset$ for all points, indicating that they are not assigned to any cluster.

Starting from each unassigned core point, the method recursively finds all its density connected points, which are assigned to the same cluster.

Some border point may be reachable from core points in more than one cluster; they may either be arbitrarily assigned to one of the clusters or to all of them (if overlapping clusters are allowed).

Those points that do not belong to any cluster are treated as outliers or noise.

Each DBSCAN cluster is a maximal connected component over the core point graph.

DBSCAN is sensitive to the choice of ϵ, in particular if clusters have different densities. The overall complexity of DBSCAN is $O(n^2)$.

DBSCAN Algorithm

DBSCAN (D, ϵ, *minpts*):

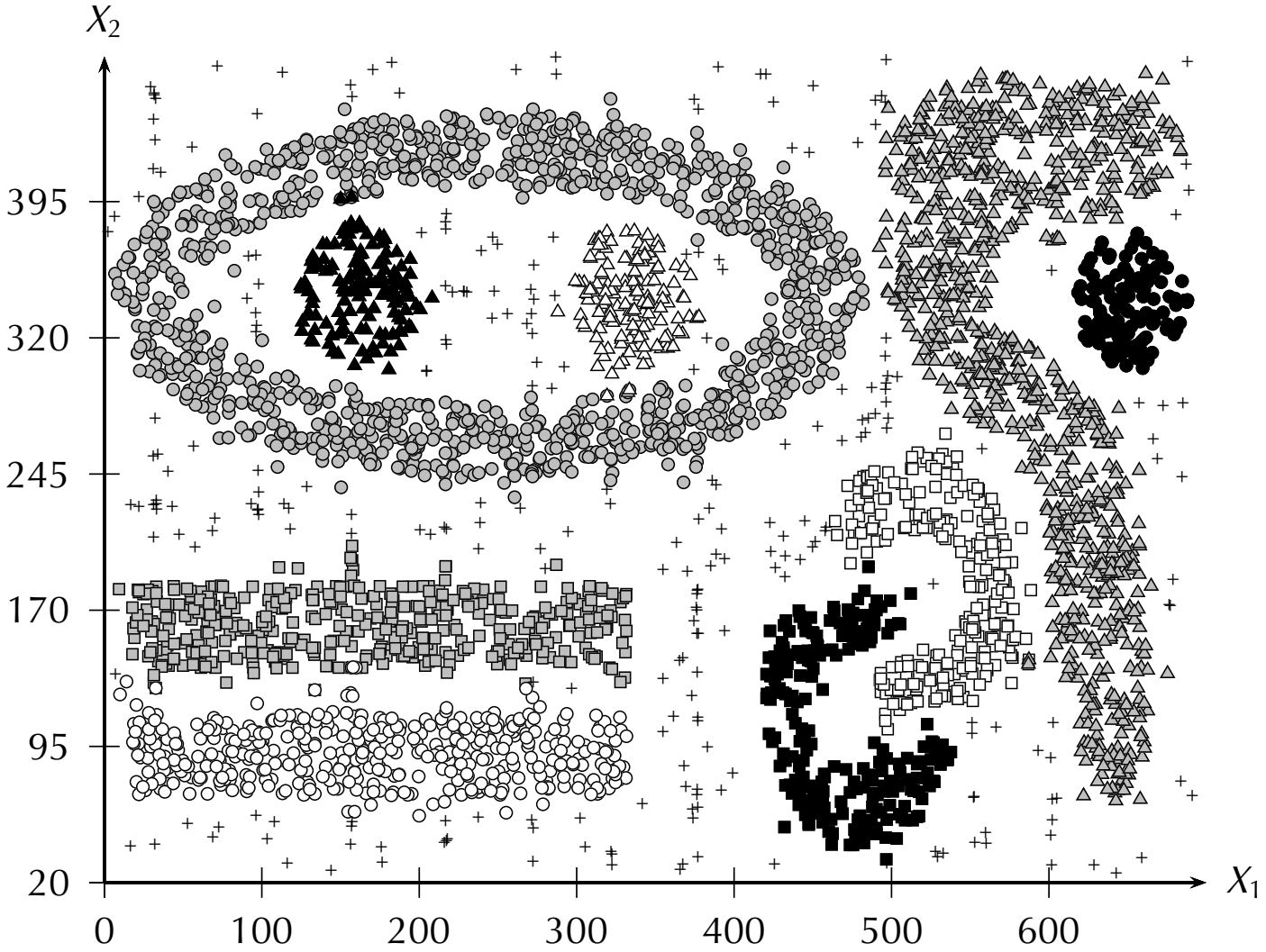
$Core \leftarrow \emptyset$
foreach $\mathbf{x}_i \in \mathbf{D}$ **do** // Find the core points
    Compute $N_\epsilon(\mathbf{x}_i)$
    $id(\mathbf{x}_i) \leftarrow \emptyset$ // cluster id for $\mathbf{x}_i$
    if $N_\epsilon(\mathbf{x}_i) \geq minpts$ **then** $Core \leftarrow Core \cup \{\mathbf{x}_i\}$
$k \leftarrow 0$ // cluster id
foreach $\mathbf{x}_i \in Core$, *such that* $id(\mathbf{x}_i) = \emptyset$ **do**
    $k \leftarrow k+1$
    $id(\mathbf{x}_i) \leftarrow k$ // assign $\mathbf{x}_i$ to cluster id k
    DENSITYCONNECTED $(\mathbf{x}_i, k)$
$\mathcal{C} \leftarrow \{C_i\}_{i=1}^{k}$, where $C_i \leftarrow \{\mathbf{x} \in \mathbf{D} \mid id(\mathbf{x}) = i\}$
$Noise \leftarrow \{\mathbf{x} \in \mathbf{D} \mid id(\mathbf{x}) = \emptyset\}$
$Border \leftarrow \mathbf{D} \setminus \{Core \cup Noise\}$
return $\mathcal{C}$, *Core*, *Border*, *Noise*

DENSITYCONNECTED (x, k):

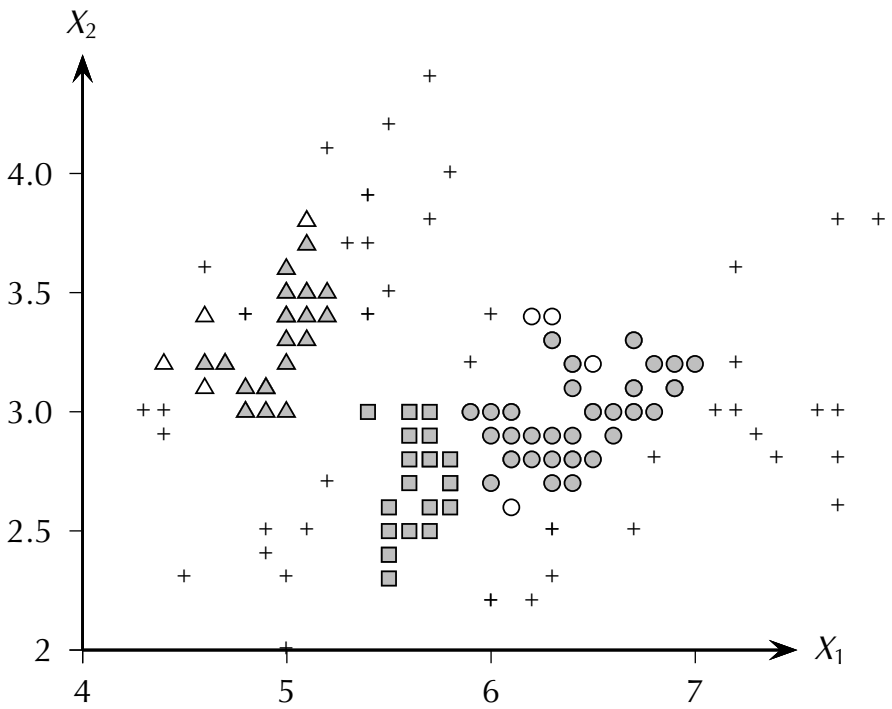
foreach $\mathbf{y} \in N_\epsilon(\mathbf{x})$ **do**
    $id(\mathbf{y}) \leftarrow k$ // assign $\mathbf{y}$ to cluster id k
    if $\mathbf{y} \in Core$ **then** DENSITYCONNECTED $(\mathbf{y}, k)$

Density-based Clusters

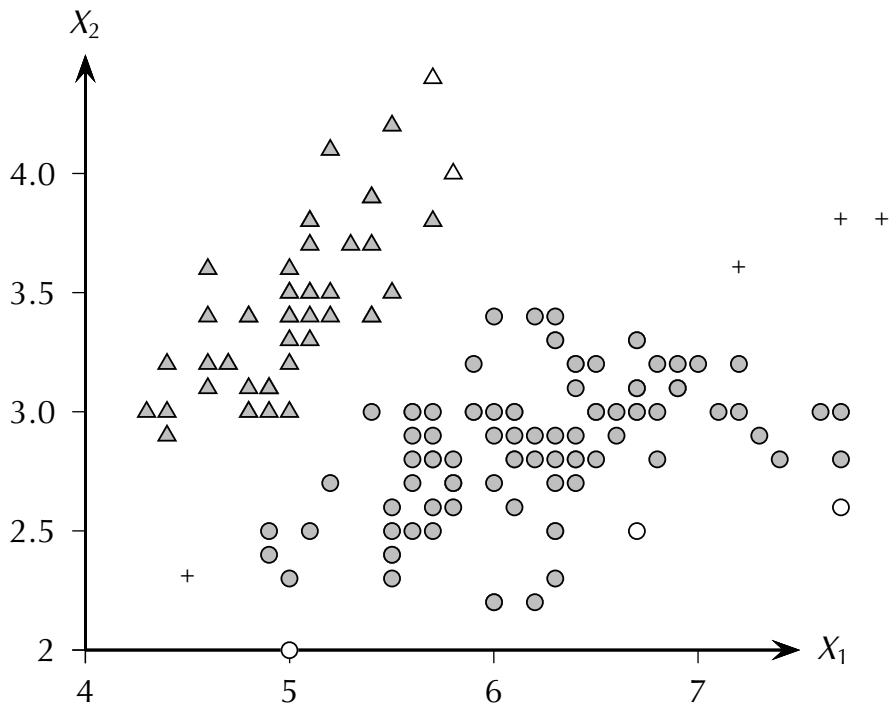
$\epsilon = 15$ and $minpts = 10$



DBSCAN Clustering: Iris Dataset



(a) $\epsilon = 0.2$, $minpts = 5$



(b) $\epsilon = 0.36$, $minpts = 3$

- Frequent Itemset Mining
- k-Means
- DBScan
- **Decision trees**

Decision Tree Classifier

Let the training dataset $\mathbf{D} = \{\mathbf{x}_i, y_i\}_{i=1}^{n}$ consist of n points in a d-dimensional space, with y_i being the class label for point $\mathbf{x}_i$.

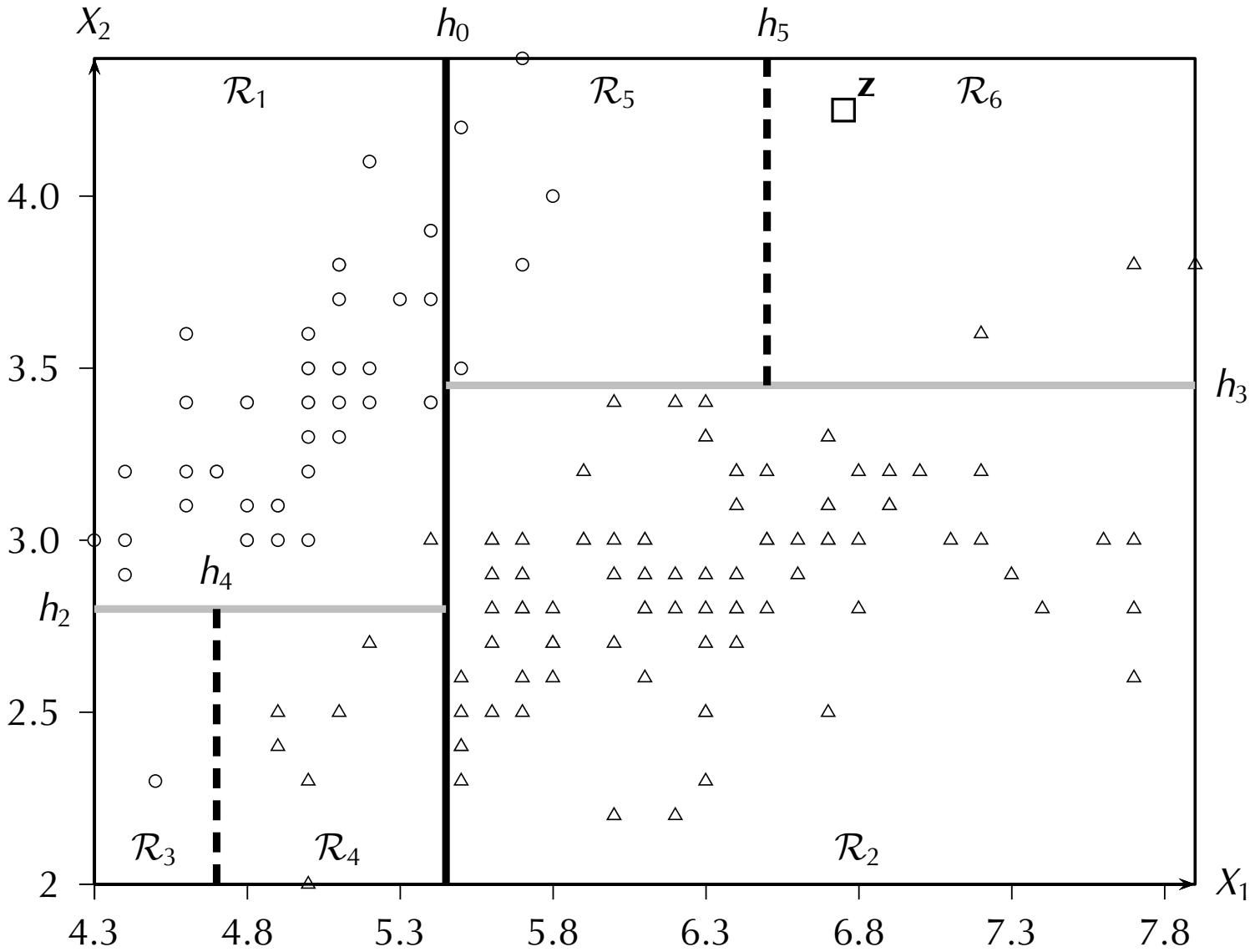
A decision tree classifier is a recursive, partition-based tree model that predicts the class $\hat{y}_i$ for each point $\mathbf{x}_i$.

Let $\mathcal{R}$ denote the data space that encompasses the set of input points $\mathbf{D}$. A decision tree uses an axis-parallel hyperplane to split the data space $\mathcal{R}$ into two resulting half-spaces or regions, say $\mathcal{R}_1$ and $\mathcal{R}_2$, which also induces a partition of the input points into $\mathbf{D}_1$ and $\mathbf{D}_2$, respectively.

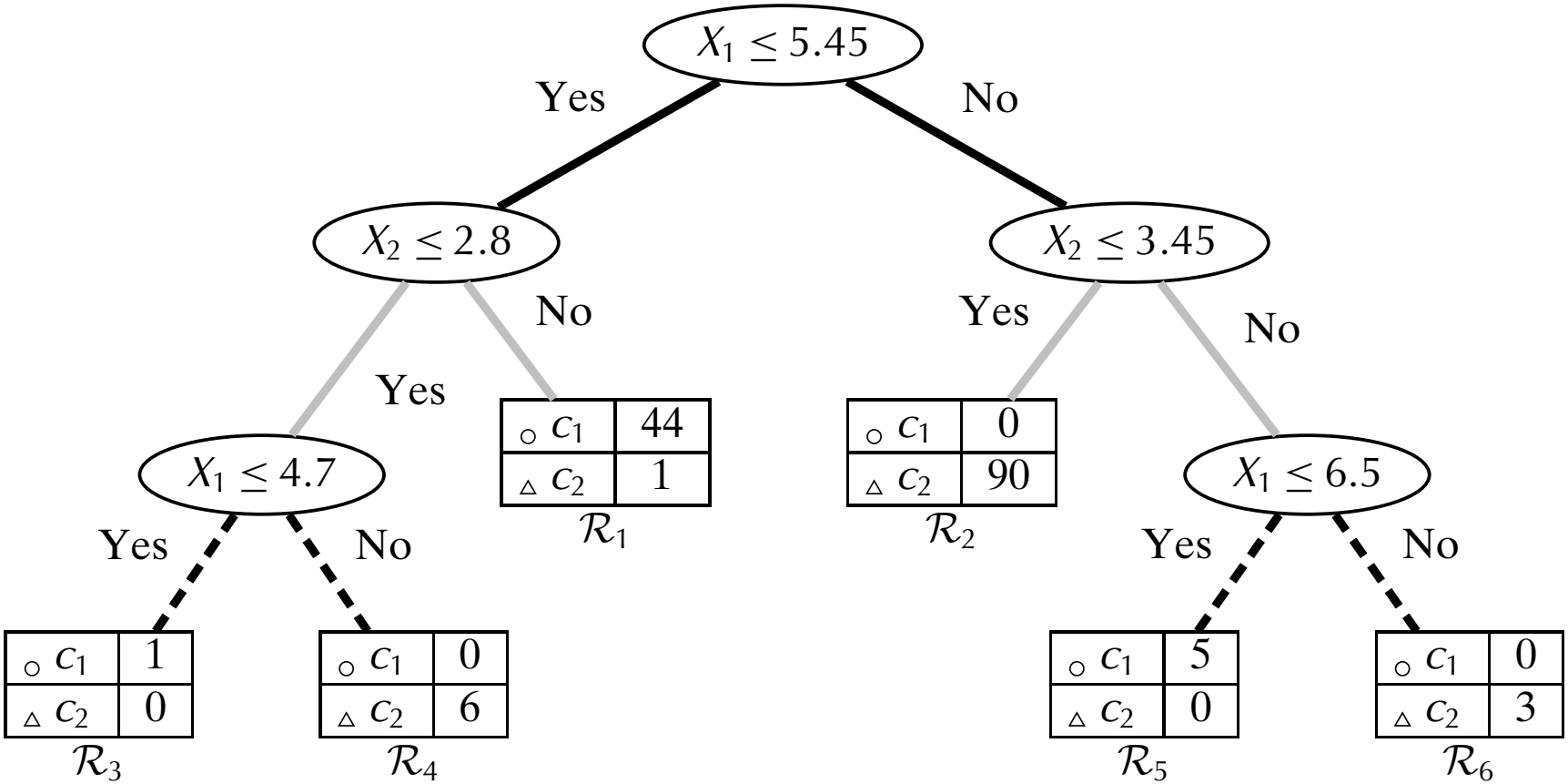
Each of these regions is recursively split via axis-parallel hyperplanes until most of the points belong to the same class.

To classify a new *test* point we have to recursively evaluate which half-space it belongs to until we reach a leaf node in the decision tree, at which point we predict its class as the label of the leaf.

Decision Tree: Recursive Splits



Decision Tree



Decision Trees: Axis-Parallel Hyperplanes

A hyperplane $h(\mathbf{x})$ is defined as the set of all points $\mathbf{x}$ that satisfy the following equation

$$h(\mathbf{x}) : \mathbf{w}^T\mathbf{x} + b = 0$$

where $\mathbf{w} \in \mathbb{R}^d$ is a *weight vector* that is normal to the hyperplane, and b is the offset of the hyperplane from the origin.

A decision tree considers only *axis-parallel hyperplanes*, that is, the weight vector must be parallel to one of the original dimensions or axes X_j:

$$h(\mathbf{x}) : x_j + b = 0$$

where the choice of the offset b yields different hyperplanes along dimension X_j.

Decision Trees: Split Points

A hyperplane specifies a decision or *split point* because it splits the data space $\mathcal{R}$ into two half-spaces. All points $\mathbf{x}$ such that $h(\mathbf{x}) \leq 0$ are on the hyperplane or to one side of the hyperplane, whereas all points such that $h(\mathbf{x}) > 0$ are on the other side.

The split point is written as $h(\mathbf{x}) \leq 0$, i.e.

$$X_j \leq v$$

where $v = -b$ is some value in the domain of attribute X_j.

The decision or split point $X_j \leq v$ thus splits the input data space $\mathcal{R}$ into two regions $\mathcal{R}_Y$ and $\mathcal{R}_N$, which denote the set of *all possible points* that satisfy the decision and those that do not.

Categorical Attributes: For a categorical attribute X_j, the split points or decisions are of the $X_j \in V$, where $V \subset dom(X_j)$, and $dom(X_j)$ denotes the domain for X_j.

Decision Trees: Data Partition and Purity

Each split of $\mathcal{R}$ into $\mathcal{R}_Y$ and $\mathcal{R}_N$ also induces a binary partition of the corresponding input data points $\mathbf{D}$. A split point of the form $X_j \leq v$ induces the data partition

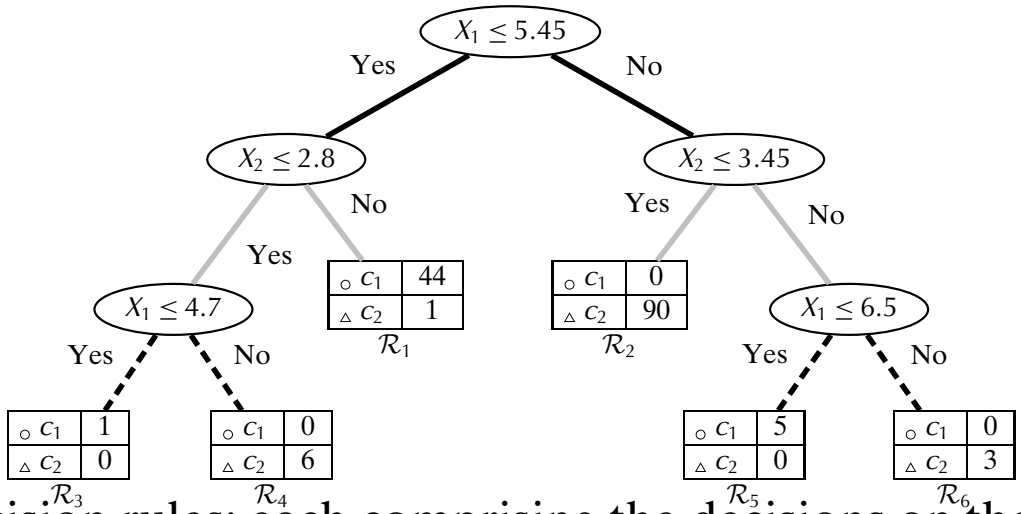
$$\mathbf{D}_Y = \{\mathbf{x} \mid \mathbf{x} \in \mathbf{D}, x_j \leq v\}$$
$$\mathbf{D}_N = \{\mathbf{x} \mid \mathbf{x} \in \mathbf{D}, x_j > v\}$$

The purity of a region $\mathcal{R}_j$ is the fraction of points with the majority label in $\mathbf{D}_j$, that is,

$$purity(\mathbf{D}_j) = \max_i \left\{ \frac{n_{ji}}{n_j} \right\}$$

where $n_j = |\mathbf{D}_j|$ is the total number of data points in the region $\mathcal{R}_j$, and n_{ji} is the number of points in $\mathbf{D}_j$ with class label c_i.

Decision Trees to Rules



A tree is a set of decision rules; each comprising the decisions on the path to a leaf:

$\mathcal{R}_3$: If $X_1 \leq 5.45$ and $X_2 \leq 2.8$ and $X_1 \leq 4.7$, then class is c_1, or

$\mathcal{R}_4$: If $X_1 \leq 5.45$ and $X_2 \leq 2.8$ and $X_1 > 4.7$, then class is c_2, or

$\mathcal{R}_1$: If $X_1 \leq 5.45$ and $X_2 > 2.8$, then class is c_1, or

$\mathcal{R}_2$: If $X_1 > 5.45$ and $X_2 \leq 3.45$, then class is c_2, or

$\mathcal{R}_5$: If $X_1 > 5.45$ and $X_2 > 3.45$ and $X_1 \leq 6.5$, then class is c_1, or

$\mathcal{R}_6$: If $X_1 > 5.45$ and $X_2 > 3.45$ and $X_1 > 6.5$, then class is c_2

Decision Tree Algorithm

The method takes as input a training dataset $\mathbf{D}$, and two parameters η and π, where η is the leaf size and π the leaf purity threshold.

Different split points are evaluated for each attribute in $\mathbf{D}$. Numeric decisions are of the form $X_j \leq v$ for some value v in the value range for attribute X_j, and categorical decisions are of the form $X_j \in V$ for some subset of values in the domain of X_j.

The best split point is chosen to partition the data into two subsets, $\mathbf{D}_Y$ and $\mathbf{D}_N$, where $\mathbf{D}_Y$ corresponds to all points $\mathbf{x} \in \mathbf{D}$ that satisfy the split decision, and $\mathbf{D}_N$ corresponds to all points that do not satisfy the split decision. The decision tree method is then called recursively on $\mathbf{D}_Y$ and $\mathbf{D}_N$.

We stop the process if the leaf size drops below η or if the purity is at least π.

Decision Tree Algorithm

DECISIONTREE $(\mathbf{D}, \eta, \pi)$:

$n \leftarrow |\mathbf{D}|$ // partition size
$n_i \leftarrow |\{\mathbf{x}_j | \mathbf{x}_j \in \mathbf{D}, y_j = c_i\}|$ // size of class c_i
$purity(\mathbf{D}) \leftarrow \max_i \left\{\frac{n_i}{n}\right\}$
if $n \leq \eta$ *or* $purity(\mathbf{D}) \geq \pi$ **then** // stopping condition
    $c^* \leftarrow \arg\max_{c_i} \left\{\frac{n_i}{n}\right\}$ // majority class
    create leaf node, and label it with class c^*
    return
$(split\ point^*, score^*) \leftarrow (\emptyset, 0)$ // initialize best split point
foreach *(attribute* X_j*)* **do**
    if *(*X_j *is numeric)* **then**
        $(v, score) \leftarrow$ EVALUATE-NUMERIC-ATTRIBUTE$(\mathbf{D}, X_j)$
        if $score > score^*$ **then** $(split\ point^*, score^*) \leftarrow (X_j \leq v, score)$
    else if *(*X_j *is categorical)* **then**
        $(V, score) \leftarrow$ EVALUATE-CATEGORICAL-ATTRIBUTE$(\mathbf{D}, X_j)$
        if $score > score^*$ **then** $(split\ point^*, score^*) \leftarrow (X_j \in V, score)$
$\mathbf{D}_Y \leftarrow \{\mathbf{x} \in \mathbf{D} \mid \mathbf{x}$ satisfies $split\ point^*\}$
$\mathbf{D}_N \leftarrow \{\mathbf{x} \in \mathbf{D} \mid \mathbf{x}$ does not satisfy $split\ point^*\}$
create internal node $split\ point^*$, with two child nodes, $\mathbf{D}_Y$ and $\mathbf{D}_N$
DECISIONTREE$(\mathbf{D}_Y)$; DECISIONTREE$(\mathbf{D}_N)$

Split Point Evaluation Measures: Entropy

Intuitively, we want to select a split point that gives the best separation or discrimination between the different class labels.

Entropy measures the amount of disorder or uncertainty in a system. A partition has lower entropy (or low disorder) if it is relatively pure, that is, if most of the points have the same label. On the other hand, a partition has higher entropy (or more disorder) if the class labels are mixed, and there is no majority class as such.

The entropy of a set of labeled points $\mathbf{D}$ is defined as follows:

$$H(\mathbf{D}) = -\sum_{i=1}^{k} P(c_i|\mathbf{D}) \log_2 P(c_i|\mathbf{D})$$

where $P(c_i|\mathbf{D})$ is the probability of class c_i in $\mathbf{D}$, and k is the number of classes.

If a region is pure, that is, has points from the same class, then the entropy is zero. On the other hand, if the classes are all mixed up, and each appears with equal probability $P(c_i|\mathbf{D}) = \frac{1}{k}$, then the entropy has the highest value, $H(\mathbf{D}) = \log_2 k$.

Split Point Evaluation Measures: Entropy

Define the *split entropy* as the weighted entropy of each of the resulting partitions

$$H(\mathbf{D}_Y, \mathbf{D}_N) = \frac{n_Y}{n} H(\mathbf{D}_Y) + \frac{n_N}{n} H(\mathbf{D}_N)$$

where $n = |\mathbf{D}|$ is the number of points in $\mathbf{D}$, and $n_Y = |\mathbf{D}_Y|$ and $n_N = |\mathbf{D}_N|$ are the number of points in $\mathbf{D}_Y$ and $\mathbf{D}_N$.

Define the *information gain* for a split point as

$$Gain(\mathbf{D}, \mathbf{D}_Y, \mathbf{D}_N) = H(\mathbf{D}) - H(\mathbf{D}_Y, \mathbf{D}_N)$$

The higher the information gain, the more the reduction in entropy, and the better the split point.

We score each split point and choose the one that gives the highest information gain.

Split Point Evaluation Measures: Gini Index and CART Measure

Gini Index: The Gini index is defined as follows:

$$G(\mathbf{D}) = 1 - \sum_{i=1}^{k} P(c_i|\mathbf{D})^2$$

If the partition is pure, then Gini index is 0.

The weighted Gini index of a split point is as follows:

$$G(\mathbf{D}_Y, \mathbf{D}_N) = \frac{n_Y}{n} G(\mathbf{D}_Y) + \frac{n_N}{n} G(\mathbf{D}_N)$$

The lower the Gini index value, the better the split point.

CART: The CART measure is

$$CART(\mathbf{D}_Y, \mathbf{D}_N) = 2 \frac{n_Y}{n} \frac{n_N}{n} \sum_{i=1}^{k} \left| P(c_i|\mathbf{D}_Y) - P(c_i|\mathbf{D}_N) \right|$$

This measure thus prefers a split point that maximizes the difference between the class probability mass function for the two partitions; the higher the CART measure, the better the split point.

Evaluating Split Points: Numeric Attributes

All of the split point evaluation measures depend on the class probability mass function (PMF) for $\mathbf{D}$, namely, $P(c_i|\mathbf{D})$, and the class PMFs for the resulting partitions $\mathbf{D}_Y$ and $\mathbf{D}_N$, namely $P(c_i|\mathbf{D}_Y)$ and $P(c_i|\mathbf{D}_N)$.

We have to evaluate split points of the form $X \leq v$. We consider only the midpoints between two successive distinct values for X in the sample $\mathbf{D}$. Let $\{v_1, \ldots, v_m\}$ denote the set of all such midpoints, such that $v_1 < v_2 < \cdots < v_m$.

For each split point $X \leq v$, we have to estimate the class PMFs:

$$\hat{P}(c_i|\mathbf{D}_Y) = \hat{P}(c_i|X \leq v)$$
$$\hat{P}(c_i|\mathbf{D}_N) = \hat{P}(c_i|X > v)$$

Using Bayes theorem, we have

$$\hat{P}(c_i|X \leq v) = \frac{\hat{P}(X \leq v|c_i)\hat{P}(c_i)}{\hat{P}(X \leq v)} = \frac{\hat{P}(X \leq v|c_i)\hat{P}(c_i)}{\sum_{j=1}^{k} \hat{P}(X \leq v|c_j)\hat{P}(c_j)}$$

Thus we have to estimate the prior probability and likelihood for each class in each partition.

Evaluating Split Points: Numeric Attributes

The prior probability for each class in $\mathbf{D}$ can be estimated as

$$\hat{P}(c_i) = \frac{1}{n}\sum_{j=1}^{n} I(y_j = c_i) = \frac{n_i}{n}$$

where y_j is the class for point $\mathbf{x}_j$, $n = |\mathbf{D}|$ is the total number of points, and n_i is the number of points in $\mathbf{D}$ with class c_i.

Define N_{vi} as the number of points $x_j \leq v$ with class c_i, where x_j is the value of data point $\mathbf{x}_j$ for the attribute X, given as

$$N_{vi} = \sum_{j=1}^{n} I(x_j \leq v \text{ and } y_j = c_i)$$

Evaluating Split Points: Numeric Attributes

We can estimate $\hat{P}(X \leq v|c_i)$ and $\hat{P}(X > v|c_i)$ as follows:

$$\hat{P}(X \leq v|c_i) = \frac{N_{vi}}{n_i}$$

$$\hat{P}(X > v|c_i) = 1 - \hat{P}(X \leq v|c_i) = \frac{n_i - N_{vi}}{n_i}$$

Finally, we have

$$\hat{P}(c_i|\mathbf{D}_Y) = \hat{P}(c_i|X \leq v) = \frac{N_{vi}}{\sum_{j=1}^{k} N_{vj}}$$

$$\hat{P}(c_i|\mathbf{D}_N) = \hat{P}(c_i|X > v) = \frac{n_i - N_{vi}}{\sum_{j=1}^{k} (n_j - N_{vj})}$$

The total cost of evaluating a numeric attribute is $O(n \log n + nk)$, where k is the number of classes, and n is the number of points.

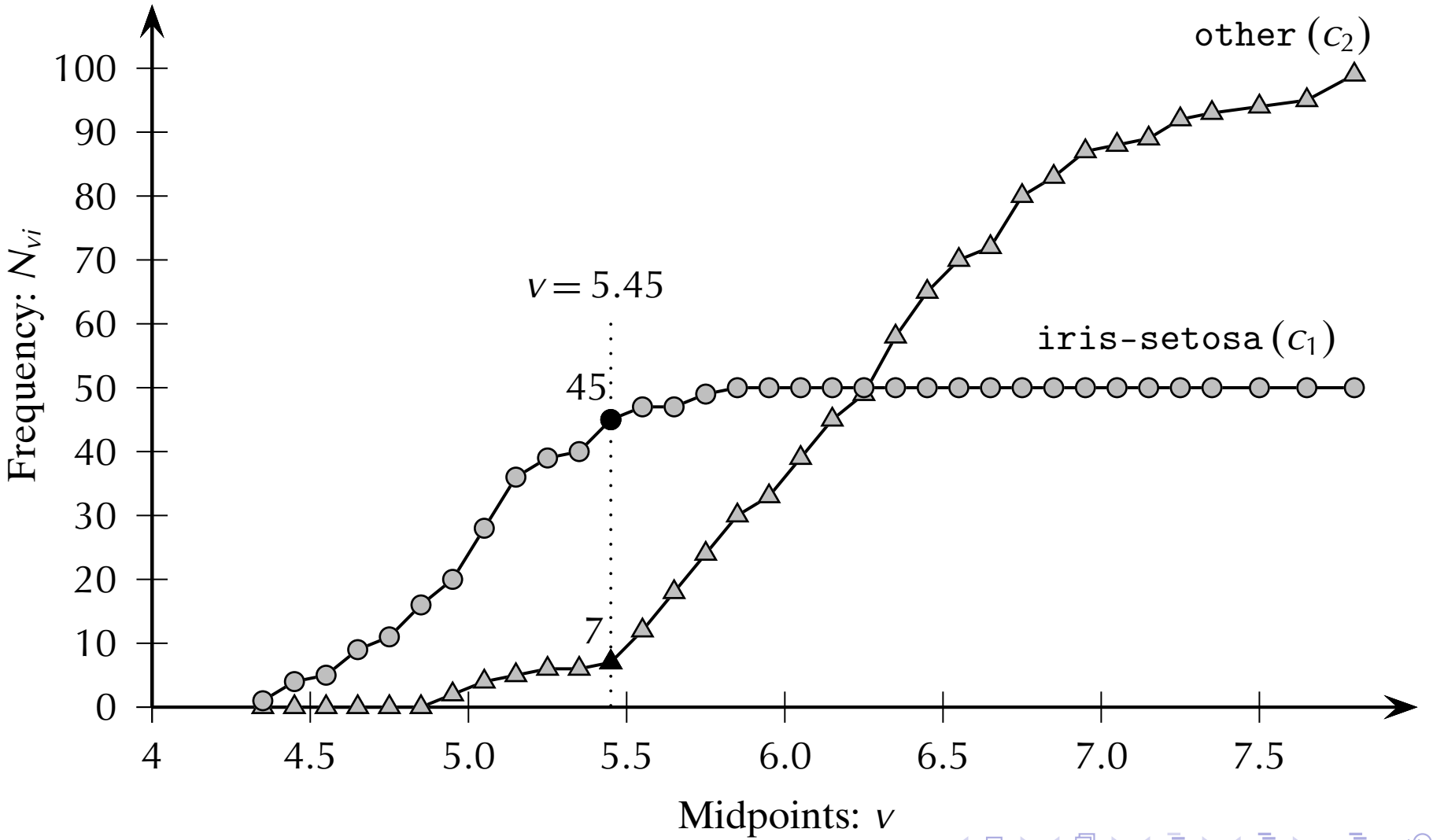
Algorithm EVALUATE-NUMERIC-ATTRIBUTE

EVALUATE-NUMERIC-ATTRIBUTE $(\mathbf{D}, X)$:

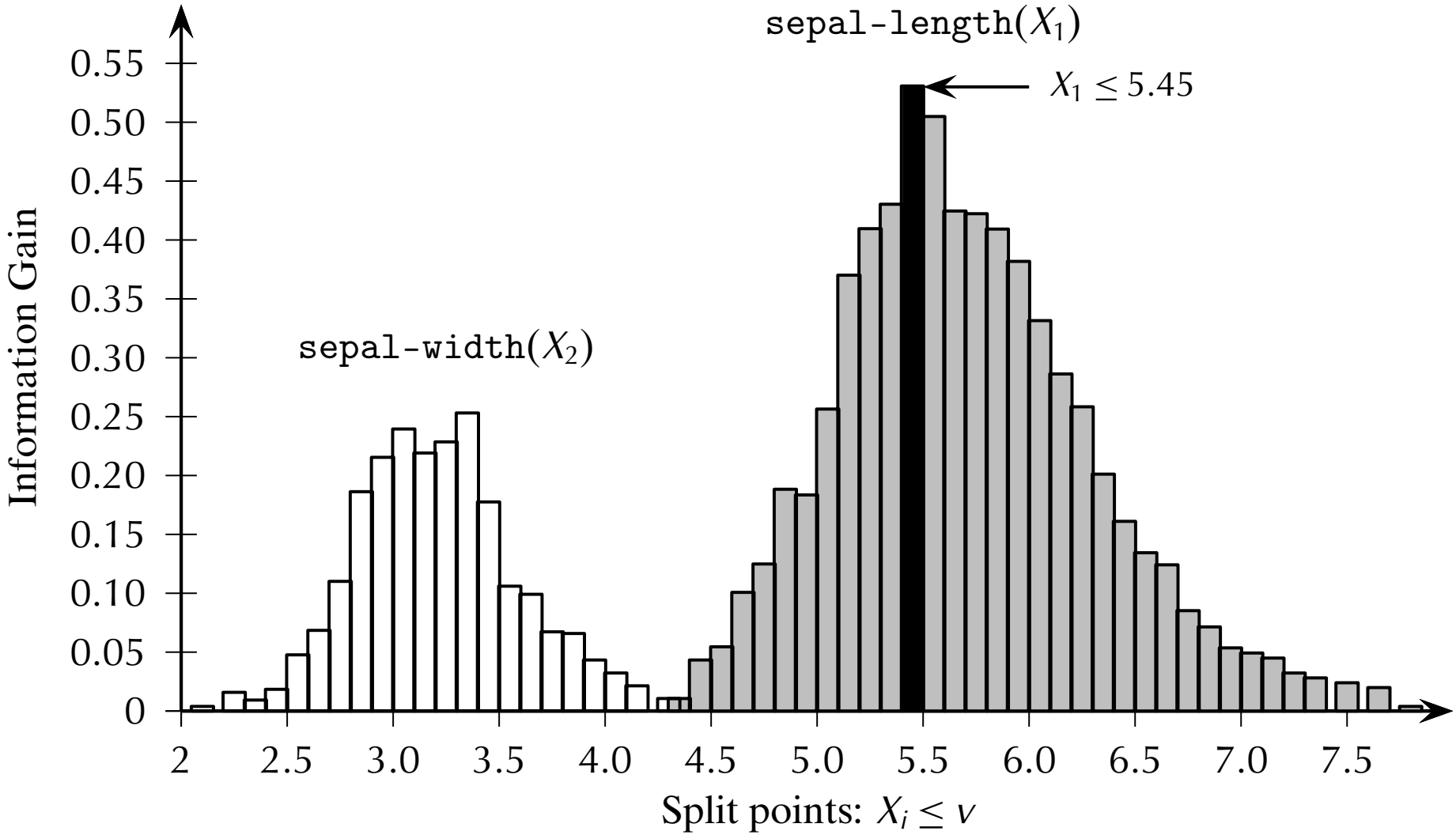
1 sort $\mathbf{D}$ on attribute X, so that $x_j \leq x_{j+1}, \forall j = 1, \ldots, n-1$
2 $\mathcal{M} \leftarrow \emptyset$ // set of midpoints
3 **for** $i = 1, \ldots, k$ **do** $n_i \leftarrow 0$
4 **for** $j = 1, \ldots, n-1$ **do**
5 **if** $y_j = c_i$ **then** $n_i \leftarrow n_i + 1$ // running count for class c_i
6 **if** $x_{j+1} \neq x_j$ **then**
7 $v \leftarrow \frac{x_{j+1} + x_j}{2}$; $\mathcal{M} \leftarrow \mathcal{M} \cup \{v\}$ // midpoints
8 **for** $i = 1, \ldots, k$ **do**
9 $N_{vi} \leftarrow n_i$ // Number of points such that $x_j \leq v$ and $y_j = c_i$
10 **if** $y_n = c_i$ **then** $n_i \leftarrow n_i + 1$
11 $v^* \leftarrow \emptyset$; $score^* \leftarrow 0$ // initialize best split point
12 **forall** $v \in \mathcal{M}$ **do**
13 **for** $i = 1, \ldots, k$ **do**
14 $\hat{P}(c_i|\mathbf{D}_Y) \leftarrow \frac{N_{vi}}{\sum_{j=1}^{k} N_{vj}}$
15 $\hat{P}(c_i|\mathbf{D}_N) \leftarrow \frac{n_i - N_{vi}}{\sum_{j=1}^{k} n_j - N_{vj}}$
16 $score(X \leq v) \leftarrow Gain(\mathbf{D}, \mathbf{D}_Y, \mathbf{D}_N)$
17 **if** $score(X \leq v) > score^*$ **then**
18 $v^* \leftarrow v$; $score^* \leftarrow score(X \leq v)$
19 **return** $(v^*, score^*)$

Iris Data: Class-specific Frequencies N_{vi}

Classes c_1 and c_2 for attribute `sepal length`



Iris Data: Information Gain for Different Splits



Categorical Attributes

For categorical X the split points are of the form $X \in V$, where $V \subset dom(X)$ and $V \neq \emptyset$. All distinct partitions of the set of values of X are considered.

If $m = |dom(X)|$, then there are $O(2^{m-1})$ distinct partitions, which can be too many. One simplification is to restrict V to be of size one, so that there are only m split points of the form $X_j \in \{v\}$, where $v \in dom(X_j)$.

Define n_{vi} as the number of points $\mathbf{x}_j \in \mathbf{D}$, with value $x_j = v$ for attribute X and having class $y_j = c_i$:

$$n_{vi} = \sum_{j=1}^{n} I(x_j = v \text{ and } y_j = c_i)$$

The class conditional empirical PMF for X is then given as

$$\hat{P}(X = v|c_i) = \frac{\hat{P}(X = v \text{ and } c_i)}{\hat{P}(c_i)} = \frac{n_{vi}}{n_i}$$

We then have

$$\hat{P}(c_i|\mathbf{D}_Y) = \frac{\sum_{v \in V} n_{vi}}{\sum_{j=1}^{k} \sum_{v \in V} n_{vj}} \qquad \hat{P}(c_i|\mathbf{D}_N) = \frac{\sum_{v \notin V} n_{vi}}{\sum_{j=1}^{k} \sum_{v \notin V} n_{vj}}$$

Algorithm EVALUATE-CATEGORICAL-ATTRIBUTE

EVALUATE-CATEGORICAL-ATTRIBUTE (**D**, X, l):

for $i = 1, \ldots, k$ **do**
  $n_i \leftarrow 0$
  forall $v \in dom(X)$ **do** $n_{vi} \leftarrow 0$
for $j = 1, \ldots, n$ **do**
  if $x_j = v$ *and* $y_j = c_i$ **then** $n_{vi} \leftarrow n_{vi} + 1$ // frequency statistics
// evaluate split points of the form $X \in V$
$V^* \leftarrow \emptyset$; $score^* \leftarrow 0$ // initialize best split point
forall $V \subset dom(X)$, *such that* $1 \leq |V| \leq l$ **do**
  for $i = 1, \ldots, k$ **do**
    $\hat{P}(c_i|\mathbf{D}_Y) \leftarrow \frac{\sum_{v \in V} n_{vi}}{\sum_{j=1}^{k} \sum_{v \in V} n_{vj}}$
    $\hat{P}(c_i|\mathbf{D}_N) \leftarrow \frac{\sum_{v \notin V} n_{vi}}{\sum_{j=1}^{k} \sum_{v \notin V} n_{vj}}$
  $score(X \in V) \leftarrow Gain(\mathbf{D}, \mathbf{D}_Y, \mathbf{D}_N)$
  if $score(X \in V) > score^*$ **then**
    $V^* \leftarrow V$; $score^* \leftarrow score(X \in V)$
return $(V^*, score^*)$

Discretized `sepal length`: Class Frequencies

Bins	v: values	Class frequencies (n_{vi})	
		c_1:`iris-setosa`	c_2:`other`
$[4.3, 5.2]$	Very Short (a_1)	39	6
$(5.2, 6.1]$	Short (a_2)	11	39
$(6.1, 7.0]$	Long (a_3)	0	43
$(7.0, 7.9]$	Very Long (a_4)	0	12

Categorical Split Points for `sepal length`

V	Split entropy	Info. gain
$\{a_1\}$	0.509	0.410
$\{a_2\}$	0.897	0.217
$\{a_3\}$	0.711	0.207
$\{a_4\}$	0.869	0.049
$\{a_1, a_2\}$	0.632	0.286
$\{a_1, a_3\}$	0.860	0.058
$\{a_1, a_4\}$	0.667	0.251
$\{a_2, a_3\}$	0.667	0.251
$\{a_2, a_4\}$	0.860	0.058
$\{a_3, a_4\}$	0.632	0.286

Best split: $X \in \{a_1\}$.

Paradigms

- Combinatorial
- **Probabilistic**
- Algebraic
- Graph-based

Domain

Models are based on one or more probability density function(s) (PDF). Given a model and a dataset, search its parameter space, which may be continuous and/or discrete.

Task

Determine the best parameter models for a dataset, according to an optimization metric.

Strategies

- Direct
- Iterative

Data: Probabilistic View

A *random variable* X is a function $X: \mathcal{O} \to \mathbb{R}$, where $\mathcal{O}$ is the set of all possible outcomes of the experiment, also called the *sample space*.

A *discrete random variable* takes on only a finite or countably infinite number of values, whereas a *continuous random variable* if it can take on any value in $\mathbb{R}$.

By default, a numeric attribute X_j is considered as the identity random variable given as

$$X(v) = v$$

for all $v \in \mathcal{O}$. Here $\mathcal{O} = \mathbb{R}$.

Discrete Variable: Long Sepal Length

Define random variable A, denoting long sepal length (7cm or more) as follows:

$$A(v) = \begin{cases} 0 & \text{if } v < 7 \\ 1 & \text{if } v \geq 7 \end{cases}$$

The sample space of A is $\mathcal{O} = [4.3, 7.9]$, and its range is $\{0, 1\}$. Thus, A is discrete.

Probability Mass Function

If X is discrete, the *probability mass function* of X is defined as

$$f(x) = P(X = x) \qquad \text{for all } x \in \mathbb{R}$$

f must obey the basic rules of probability. That is, f must be non-negative:

$$f(x) \geq 0$$

and the sum of all probabilities should add to 1:

$$\sum_x f(x) = 1$$

Intuitively, for a discrete variable X, the probability is concentrated or massed at only discrete values in the range of X, and is zero for all other values.

Sepal Length: Bernoulli Distribution

Iris Dataset Extract: `sepal length` (in centimeters)

5.9	6.9	6.6	4.6	6.0	4.7	6.5	5.8	6.7	6.7	5.1	5.1	5.7	6.1	4.9
5.0	5.0	5.7	5.0	7.2	5.9	6.5	5.7	5.5	4.9	5.0	5.5	4.6	7.2	6.8
5.4	5.0	5.7	5.8	5.1	5.6	5.8	5.1	6.3	6.3	5.6	6.1	6.8	7.3	5.6
4.8	7.1	5.7	5.3	5.7	5.7	5.6	4.4	6.3	5.4	6.3	6.9	7.7	6.1	5.6
6.1	6.4	5.0	5.1	5.6	5.4	5.8	4.9	4.6	5.2	7.9	7.7	6.1	5.5	4.6
4.7	4.4	6.2	4.8	6.0	6.2	5.0	6.4	6.3	6.7	5.0	5.9	6.7	5.4	6.3
4.8	4.4	6.4	6.2	6.0	7.4	4.9	7.0	5.5	6.3	6.8	6.1	6.5	6.7	6.7
4.8	4.9	6.9	4.5	4.3	5.2	5.0	6.4	5.2	5.8	5.5	7.6	6.3	6.4	6.3
5.8	5.0	6.7	6.0	5.1	4.8	5.7	5.1	6.6	6.4	5.2	6.4	7.7	5.8	4.9
5.4	5.1	6.0	6.5	5.5	7.2	6.9	6.2	6.5	6.0	5.4	5.5	6.7	7.7	5.1

Define random variable A as follows: $A(v) = \begin{cases} 0 & \text{if } v < 7 \\ 1 & \text{if } v \geq 7 \end{cases}$

We find that only 13 Irises have sepal length of at least 7 cm. Thus, the probability mass function of A can be estimated as:

$$f(1) = P(A = 1) = \frac{13}{150} = 0.087 = p$$

and

$$f(0) = P(A = 0) = \frac{137}{150} = 0.913 = 1 - p$$

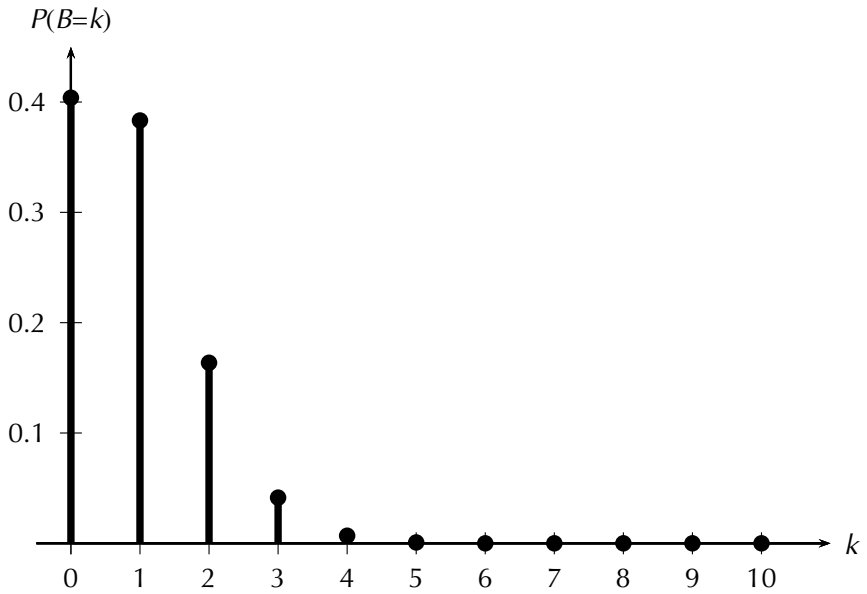
A has a *Bernoulli distribution* with parameter $p \in [0, 1]$, which denotes the probability of a *success*, that is, the probability of picking an Iris with a long sepal length at random from the set of all points.

Sepal Length: Binomial Distribution

Define discrete random variable B, denoting the number of Irises with long sepal length in m independent Bernoulli trials with probability of success p. In this case, B takes on the discrete values $[0, m]$, and its probability mass function is given by the *Binomial distribution*

$$f(k) = P(B = k) = \binom{m}{k} p^k (1-p)^{m-k}$$

Binomial distribution for long sepal length ($p = 0.087$) for $m = 10$ trials



Probability Density Function

If X is continuous, the *probability density function* of X is defined as

$$P\big(X \in [a, b]\big) = \int_a^b f(x)\ dx$$

f must obey the basic rules of probability. That is, f must be non-negative:

$$f(x) \geq 0$$

and the sum of all probabilities should add to 1:

$$\int_{-\infty}^{\infty} f(x)\ dx = 1$$

Note that $P(X = v) = 0$ for all $v \in \mathbb{R}$ since there are infinite possible values in the sample space. What it means is that the probability mass is spread so thinly over the range of values that it can be measured only over intervals $[a, b] \subset \mathbb{R}$, rather than at specific points.

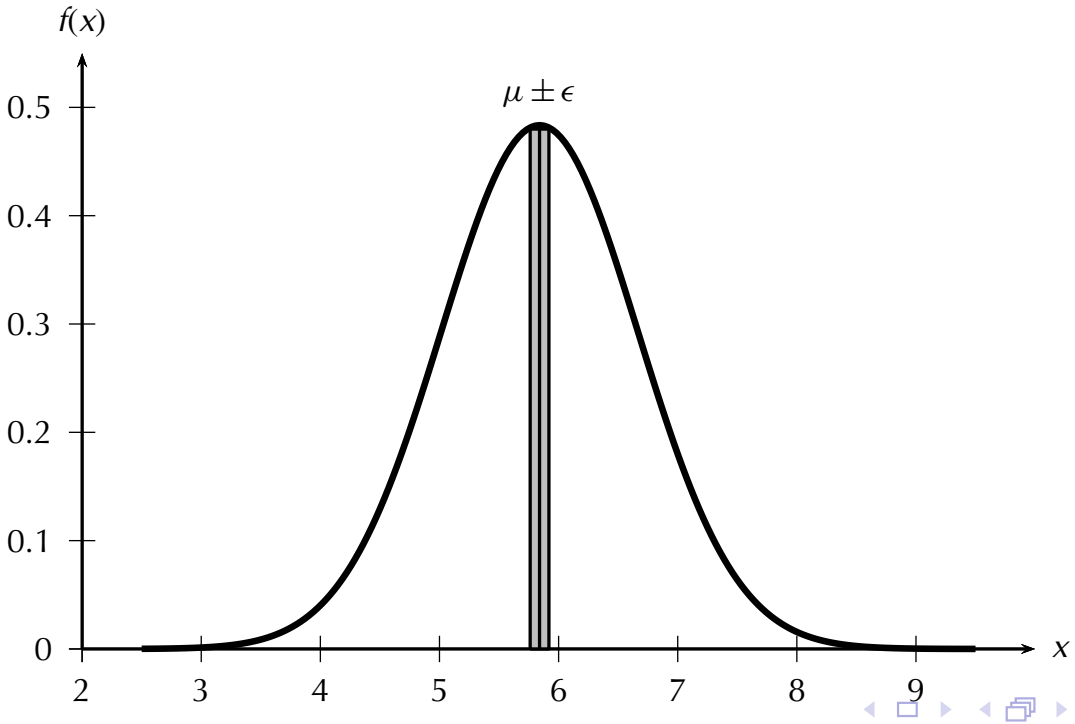
Sepal Length: Normal Distribution

We model sepal length via the *Gaussian* or *normal* density function, given as

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{\frac{-(x-\mu)^2}{2\sigma^2}\right\}$$

where $\mu = \frac{1}{n}\sum_{i=1}^{n} x_i$ is the mean value, and $\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2$ is the variance.

Normal distribution for sepal length: $\mu = 5.84$, $\sigma^2 = 0.681$



Cumulative Distribution Function

For random variable X, its *cumulative distribution function (CDF)*
$F : \mathbb{R} \to [0, 1]$, gives the probability of observing a value at most some given value x:

$$F(x) = P(X \leq x) \qquad \text{for all } -\infty < x < \infty$$

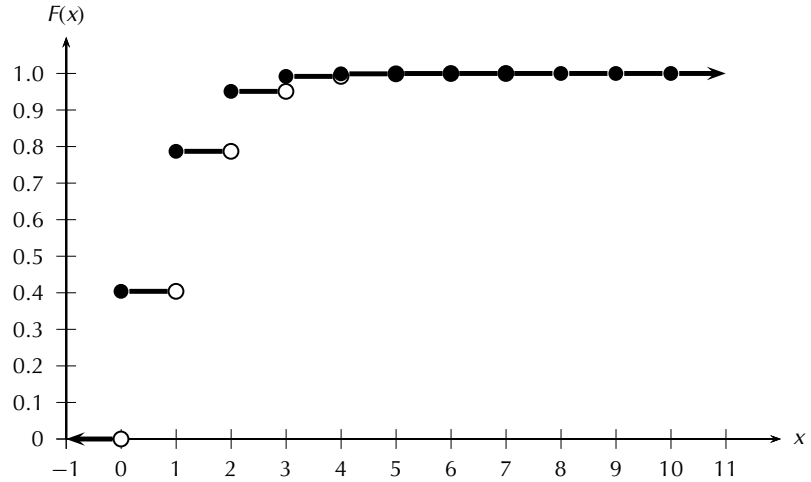
When X is discrete, F is given as

$$F(x) = P(X \leq x) = \sum_{u \leq x} f(u)$$

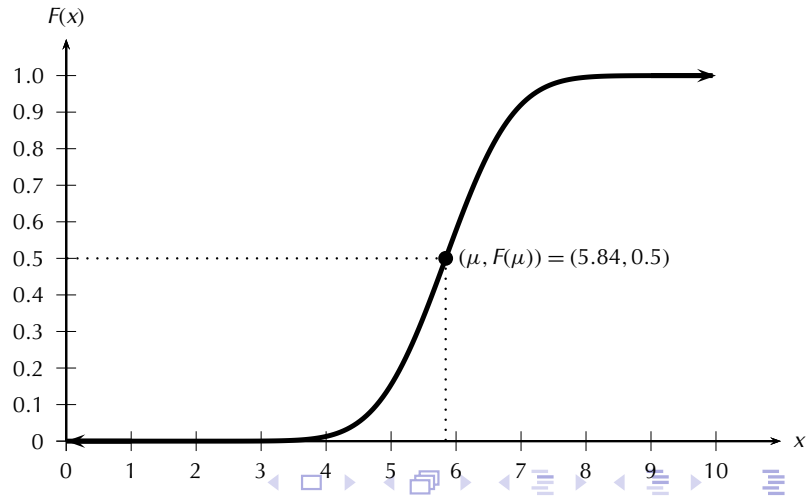
When X is continuous, F is given as

$$F(x) = P(X \leq x) = \int_{-\infty}^{x} f(u)\, du$$

CDF for binomial distribution
$(p = 0.087, m = 10)$



CDF for the normal distribution
$(\mu = 5.84, \sigma^2 = 0.681)$



Bivariate Random Variable: Joint Probability Mass Function

Define discrete random variables

$$\text{long sepal length: } X_1(v) = \begin{cases} 1 & \text{if } v \geq 7 \\ 0 & \text{otherwise} \end{cases}$$

$$\text{long sepal width: } X_2(v) = \begin{cases} 1 & \text{if } v \geq 3.5 \\ 0 & \text{otherwise} \end{cases}$$

The bivariate random variable

$$\mathbf{X} = \begin{pmatrix} X_1 \\ X_2 \end{pmatrix}$$

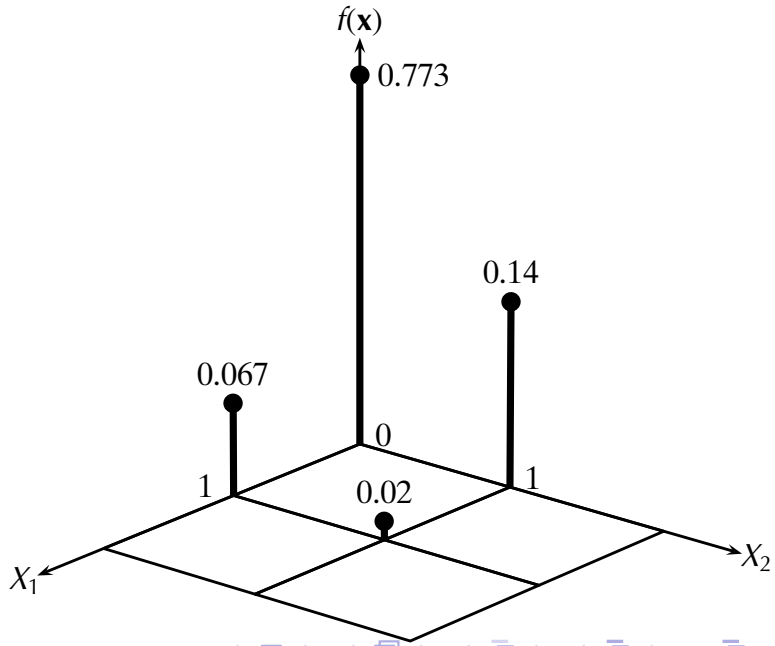
has the joint probability mass function

$$f(\mathbf{x}) = P(\mathbf{X} = \mathbf{x})$$

$$\text{i.e., } f(x_1, x_2) = P(X_1 = x_1, X_2 = x_2)$$

Iris: joint PMF for long sepal length and sepal width

$$f(0,0) = P(X_1 = 0, X_2 = 0) = 116/150 = 0.773$$
$$f(0,1) = P(X_1 = 0, X_2 = 1) = 21/150 = 0.140$$
$$f(1,0) = P(X_1 = 1, X_2 = 0) = 10/150 = 0.067$$
$$f(1,1) = P(X_1 = 1, X_2 = 1) = 3/150 = 0.020$$



Bivariate Random Variable: Probability Density Function

Bivariate Normal: modeling joint distribution for long sepal length (X_1) and sepal width (X_2)

$$f(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{2\pi\sqrt{|\boldsymbol{\Sigma}|}} \exp\left\{-\frac{(\mathbf{x}-\boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1} (\mathbf{x}-\boldsymbol{\mu})}{2}\right\}$$

where $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ specify the 2D mean and covariance matrix:

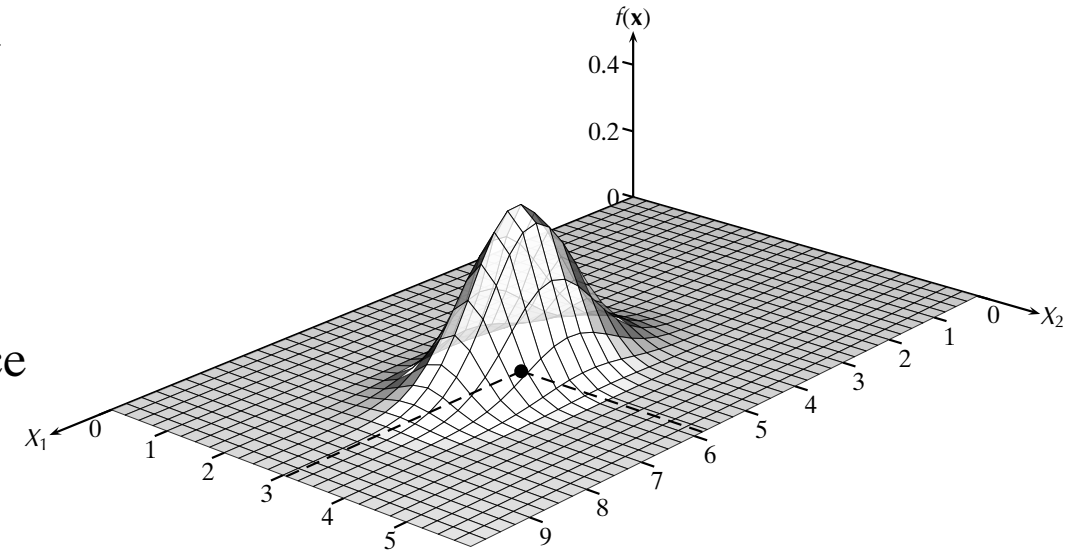
$$\boldsymbol{\mu} = (\mu_1, \mu_2)^T \qquad \boldsymbol{\Sigma} = \begin{pmatrix} \sigma_1^2 & \sigma_{12} \\ \sigma_{21} & \sigma_2^2 \end{pmatrix}$$

with mean $\mu_i = \frac{1}{n}\sum_{k=1}^{n} x_{ki}$ and covariance $\sigma_{ij} = \frac{1}{n}\sum_{k=1}(x_{ki} - \mu_i)(x_{kj} - \mu_j)$. Also, $\sigma_i^2 = \sigma_{ii}$.

Bivariate Normal

$$\boldsymbol{\mu} = (5.843, 3.054)^T$$

$$\boldsymbol{\Sigma} = \begin{pmatrix} 0.681 & -0.039 \\ -0.039 & 0.187 \end{pmatrix}$$



- **Expectation-Maximization**
- DenClue
- Naive Bayes

Expectation-Maximization Clustering: Gaussian Mixture Model

Let X_a denote the random variable corresponding to the ath attribute. Let $\mathbf{X} = (X_1, X_2, \ldots, X_d)$ denote the vector random variable across the d-attributes, with $\mathbf{x}_j$ being a data sample from $\mathbf{X}$.

We assume that each cluster C_i is characterized by a multivariate normal distribution

$$f_i(\mathbf{x}) = f(\mathbf{x}|\boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i) = \frac{1}{(2\pi)^{\frac{d}{2}}|\boldsymbol{\Sigma}_i|^{\frac{1}{2}}} \exp\left\{-\frac{(\mathbf{x}-\boldsymbol{\mu}_i)^T\boldsymbol{\Sigma}_i^{-1}(\mathbf{x}-\boldsymbol{\mu}_i)}{2}\right\}$$

where the cluster mean $\boldsymbol{\mu}_i \in \mathbb{R}^d$ and covariance matrix $\boldsymbol{\Sigma}_i \in \mathbb{R}^{d\times d}$ are both unknown parameters.

The probability density function of $\mathbf{X}$ is given as a *Gaussian mixture model* over all the k clusters

$$f(\mathbf{x}) = \sum_{i=1}^{k} f_i(\mathbf{x})P(C_i) = \sum_{i=1}^{k} f(\mathbf{x}|\boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i)P(C_i)$$

where the prior probabilities $P(C_i)$ are called the *mixture parameters*, which must satisfy the condition

$$\sum_{i=1}^{k} P(C_i) = 1$$

Expectation-Maximization Clustering: Maximum Likelihood Estimation

We write the set of all the model parameters compactly as

$$\boldsymbol{\theta} = \left\{\boldsymbol{\mu}_1, \boldsymbol{\Sigma}_1, P(C_i) \ldots, \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k, P(C_k)\right\}$$

Given the dataset $\mathbf{D}$, we define the *likelihood* of $\boldsymbol{\theta}$ as the conditional probability of the data $\mathbf{D}$ given the model parameters $\boldsymbol{\theta}$

$$P(\mathbf{D}|\boldsymbol{\theta}) = \prod_{j=1}^{n} f(\mathbf{x}_j)$$

The goal of maximum likelihood estimation (MLE) is to choose the parameters $\boldsymbol{\theta}$ that maximize the likelihood. We do this by maximizing the log of the likelihood function

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}}\{\ln P(\mathbf{D}|\boldsymbol{\theta})\}$$

where the *log-likelihood* function is given as

$$\ln P(\mathbf{D}|\boldsymbol{\theta}) = \sum_{j=1}^{n} \ln f(\mathbf{x}_j) = \sum_{j=1}^{n} \ln\left(\sum_{i=1}^{k} f(\mathbf{x}_j|\boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i) P(C_i)\right)$$

Expectation-Maximization Clustering

Directly maximizing the log-likelihood over $\boldsymbol{\theta}$ is hard. Instead, we can use the expectation-maximization (EM) approach for finding the maximum likelihood estimates for the parameters $\boldsymbol{\theta}$.

EM is a two-step iterative approach that starts from an initial guess for the parameters $\boldsymbol{\theta}$. Given the current estimates for $\boldsymbol{\theta}$, in the *expectation step* EM computes the cluster posterior probabilities $P(C_i|\mathbf{x}_j)$ via the Bayes theorem:

$$P(C_i|\mathbf{x}_j) = \frac{P(C_i \text{ and } \mathbf{x}_j)}{P(\mathbf{x}_j)} = \frac{P(\mathbf{x}_j|C_i)P(C_i)}{\sum_{a=1}^{k} P(\mathbf{x}_j|C_a)P(C_a)} = \frac{f_i(\mathbf{x}_j) \cdot P(C_i)}{\sum_{a=1}^{k} f_a(\mathbf{x}_j) \cdot P(C_a)}$$

In the *maximization step*, using the weights $P(C_i|\mathbf{x}_j)$ EM re-estimates $\boldsymbol{\theta}$, that is, it re-estimates the parameters $\boldsymbol{\mu}_i$, $\boldsymbol{\Sigma}_i$, and $P(C_i)$ for each cluster C_i. The re-estimated mean is given as the weighted average of all the points, the re-estimated covariance matrix is given as the weighted covariance over all pairs of dimensions, and the re-estimated prior probability for each cluster is given as the fraction of weights that contribute to that cluster.

EM in One Dimension: Expectation Step

Let **D** comprise of a single attribute X, with each point $x_j \in \mathbb{R}$ a random sample from X. For the mixture model, we use univariate normals for each cluster:

$$f_i(x) = f(x|\mu_i, \sigma_i^2) = \frac{1}{\sqrt{2\pi}\sigma_i} \exp\left\{-\frac{(x-\mu_i)^2}{2\sigma_i^2}\right\}$$

with the cluster parameters μ_i, σ_i^2, and $P(C_i)$.

Initialization: For each cluster C_i, with $i = 1, 2, \ldots, k$, we can randomly initialize the cluster parameters μ_i, σ_i^2, and $P(C_i)$.

Expectation Step: Given the mean μ_i, variance σ_i^2, and prior probability $P(C_i)$ for each cluster, the cluster posterior probability is computed as

$$w_{ij} = P(C_i|x_j) = \frac{f(x_j|\mu_i, \sigma_i^2) \cdot P(C_i)}{\sum_{a=1}^{k} f(x_j|\mu_a, \sigma_a^2) \cdot P(C_a)}$$

EM in One Dimension: Maximization Step

Given w_{ij} values, the re-estimated cluster mean is

$$\mu_i = \frac{\sum_{j=1}^{n} w_{ij} \cdot x_j}{\sum_{j=1}^{n} w_{ij}}$$

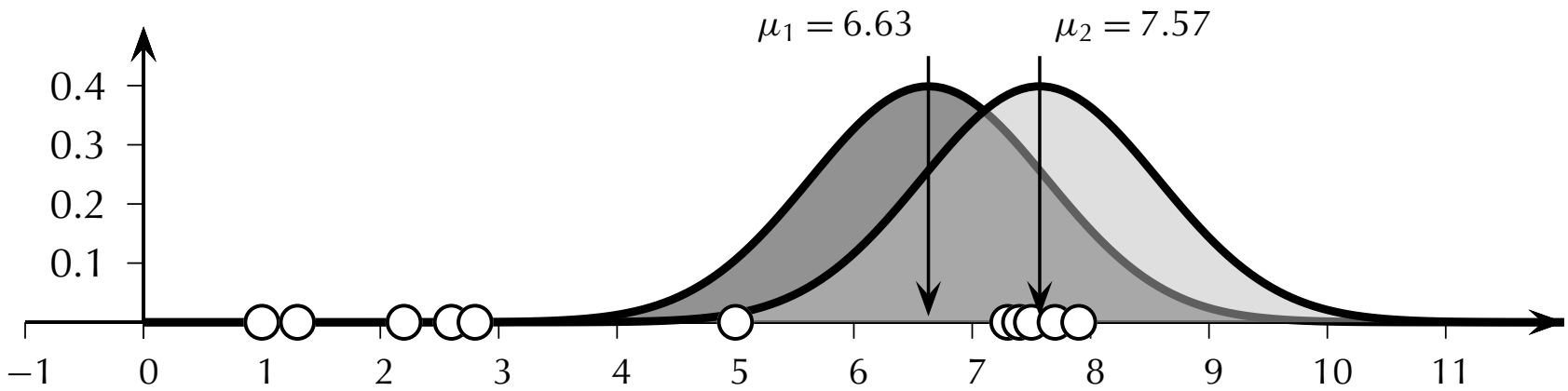
The re-estimated value of the cluster variance is computed as the weighted variance across all the points:

$$\sigma_i^2 = \frac{\sum_{j=1}^{n} w_{ij}(x_j - \mu_i)^2}{\sum_{j=1}^{n} w_{ij}}$$

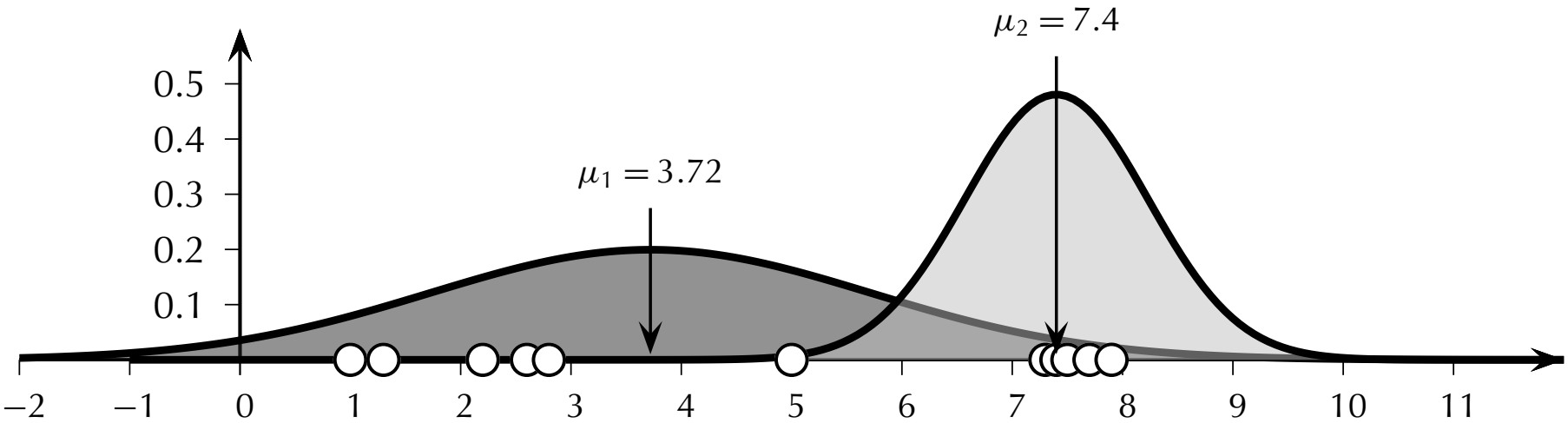
The prior probability of cluster C_i is re-estimated as

$$P(C_i) = \frac{\sum_{j=1}^{n} w_{ij}}{n}$$

EM in One Dimension

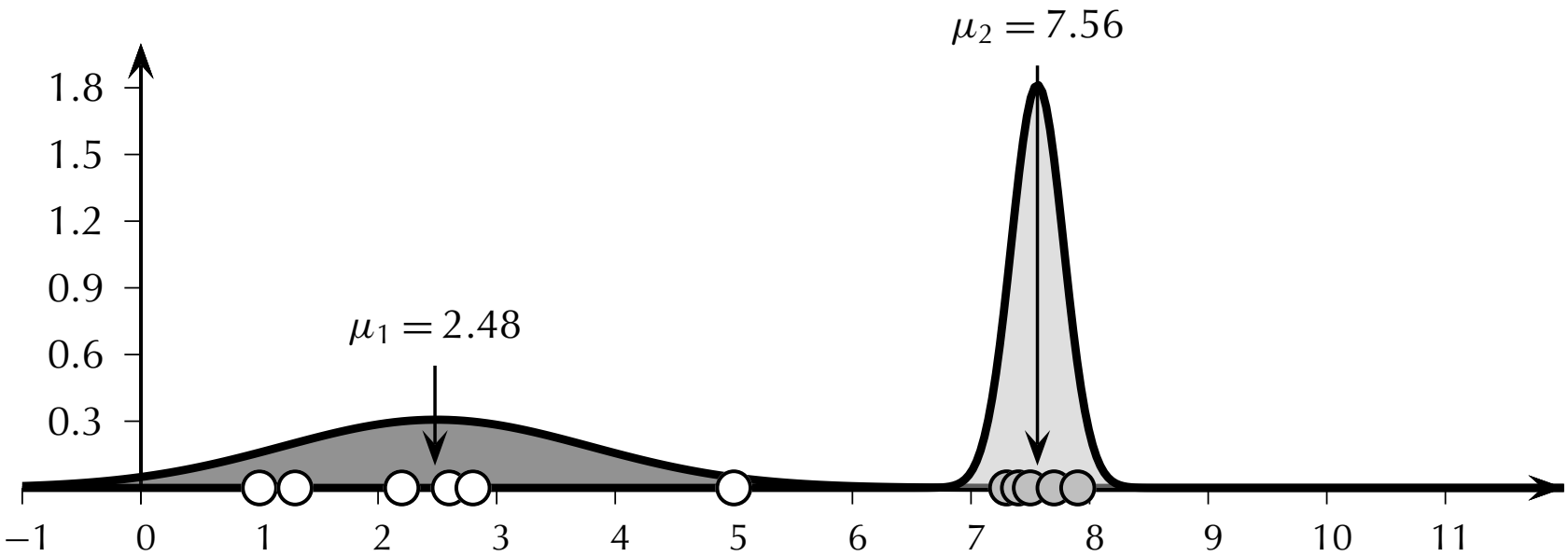


(a) Initialization: $t = 0$



(b) Iteration: $t = 1$

EM in One Dimension: Final Clusters



(c) Iteration: $t = 5$ (converged)

EM in d Dimensions

Each cluster we have to reestiamte the $d \times d$ covariance matrix:

$$\mathbf{\Sigma}_i = \begin{pmatrix} (\sigma_1^i)^2 & \sigma_{12}^i & \dots & \sigma_{1d}^i \\ \sigma_{21}^i & (\sigma_2^i)^2 & \dots & \sigma_{2d}^i \\ \vdots & \vdots & \ddots & \\ \sigma_{d1}^i & \sigma_{d2}^i & \dots & (\sigma_d^i)^2 \end{pmatrix}$$

It requires $O(d^2)$ parameters, which may be too many for reliable estimation. A simplification is to assume that all dimensions are independent, which leads to a diagonal covariance matrix:

$$\mathbf{\Sigma}_i = \begin{pmatrix} (\sigma_1^i)^2 & 0 & \dots & 0 \\ 0 & (\sigma_2^i)^2 & \dots & 0 \\ \vdots & \vdots & \ddots & \\ 0 & 0 & \dots & (\sigma_d^i)^2 \end{pmatrix}$$

EM in d Dimensions

Expectation Step: Given $\boldsymbol{\mu}_i$, $\boldsymbol{\Sigma}_i$, and $P(C_i)$, the posterior probability is given as

$$w_{ij} = P(C_i|\mathbf{x}_j) = \frac{f_i(\mathbf{x}_j) \cdot P(C_i)}{\sum_{a=1}^{k} f_a(\mathbf{x}_j) \cdot P(C_a)}$$

Maximization Step: Given the weights w_{ij}, in the maximization step, we re-estimate $\boldsymbol{\Sigma}_i$, $\boldsymbol{\mu}_i$ and $P(C_i)$.
The mean $\boldsymbol{\mu}_i$ for cluster C_i can be estimated as

$$\boldsymbol{\mu}_i = \frac{\sum_{j=1}^{n} w_{ij} \cdot \mathbf{x}_j}{\sum_{j=1}^{n} w_{ij}}$$

The covariance matrix $\boldsymbol{\Sigma}_i$ is re-estimated via the outer-product form

$$\boldsymbol{\Sigma}_i = \frac{\sum_{j=1}^{n} w_{ij}(\mathbf{x}_j - \boldsymbol{\mu}_i)(\mathbf{x}_j - \boldsymbol{\mu}_i)^T}{\sum_{j=1}^{n} w_{ij}}$$

The prior probability $P(C_i)$ for each cluster is

$$P(C_i) = \frac{\sum_{j=1}^{n} w_{ij}}{n}$$

Expectation-Maximization Clustering Algorithm

EXPECTATION-MAXIMIZATION (D, k, ϵ):

1 $t \leftarrow 0$

2 Randomly initialize $\boldsymbol{\mu}_1^t, \ldots, \boldsymbol{\mu}_k^t$

3 $\boldsymbol{\Sigma}_i^t \leftarrow \mathbf{I}, \ \forall i = 1, \ldots, k$

4 **repeat**

5 $\quad t \leftarrow t+1$

6 $\quad$ **for** $i = 1, \ldots, k$ *and* $j = 1, \ldots, n$ **do**

7 $\quad\quad w_{ij} \leftarrow \frac{f(\mathbf{x}_j|\boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i) \cdot P(C_i)}{\sum_{a=1}^{k} f(\mathbf{x}_j|\boldsymbol{\mu}_a, \boldsymbol{\Sigma}_a) \cdot P(C_a)}$ // posterior probability $P^t(C_i|\mathbf{x}_j)$

8 $\quad$ **for** $i = 1, \ldots, k$ **do**

9 $\quad\quad \boldsymbol{\mu}_i^t \leftarrow \frac{\sum_{j=1}^{n} w_{ij} \cdot \mathbf{x}_j}{\sum_{j=1}^{n} w_{ij}}$ // re-estimate mean

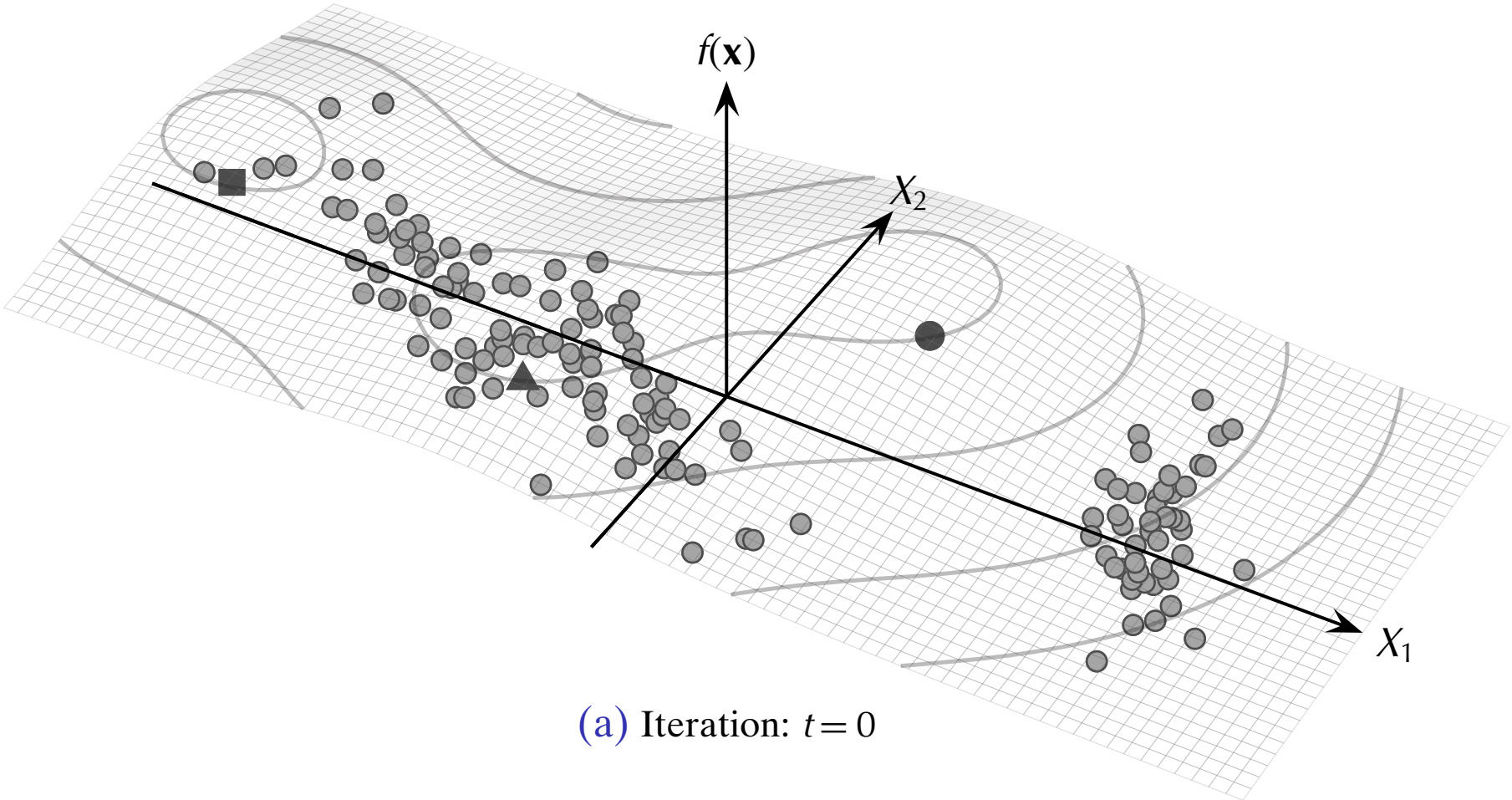
10 $\quad\quad \boldsymbol{\Sigma}_i^t \leftarrow \frac{\sum_{j=1}^{n} w_{ij} (\mathbf{x}_j - \boldsymbol{\mu}_i)(\mathbf{x}_j - \boldsymbol{\mu}_i)^T}{\sum_{j=1}^{n} w_{ij}}$ // re-estimate covariance matrix

11 $\quad\quad P^t(C_i) \leftarrow \frac{\sum_{j=1}^{n} w_{ij}}{n}$ // re-estimate priors

12 **until** $\sum_{i=1}^{k} \left\| \boldsymbol{\mu}_i^t - \boldsymbol{\mu}_i^{t-1} \right\|^2 \leq \epsilon$

EM Clustering in 2D

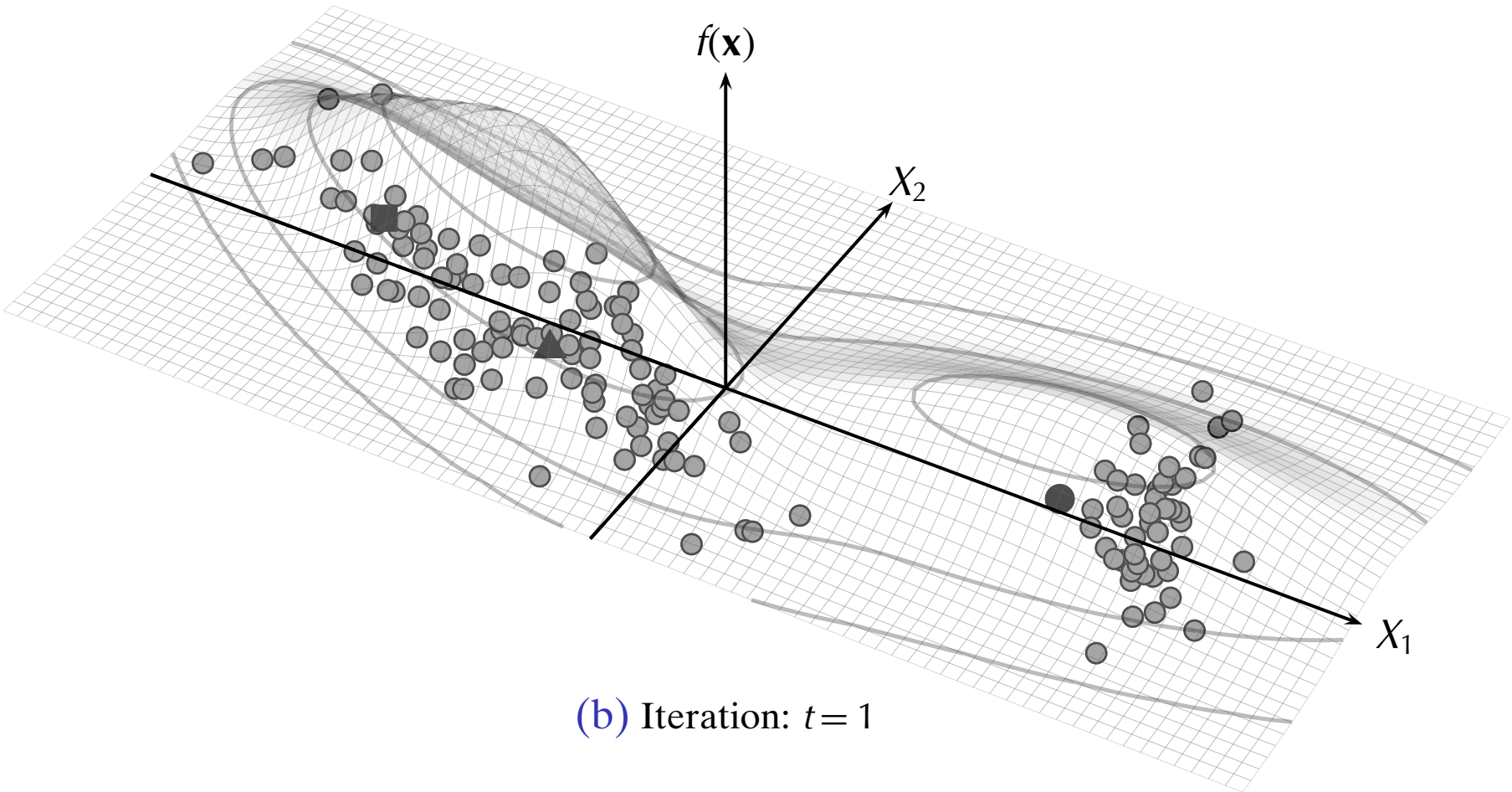
Mixture of $k = 3$ Gaussians



(a) Iteration: $t = 0$

EM Clustering in 2D

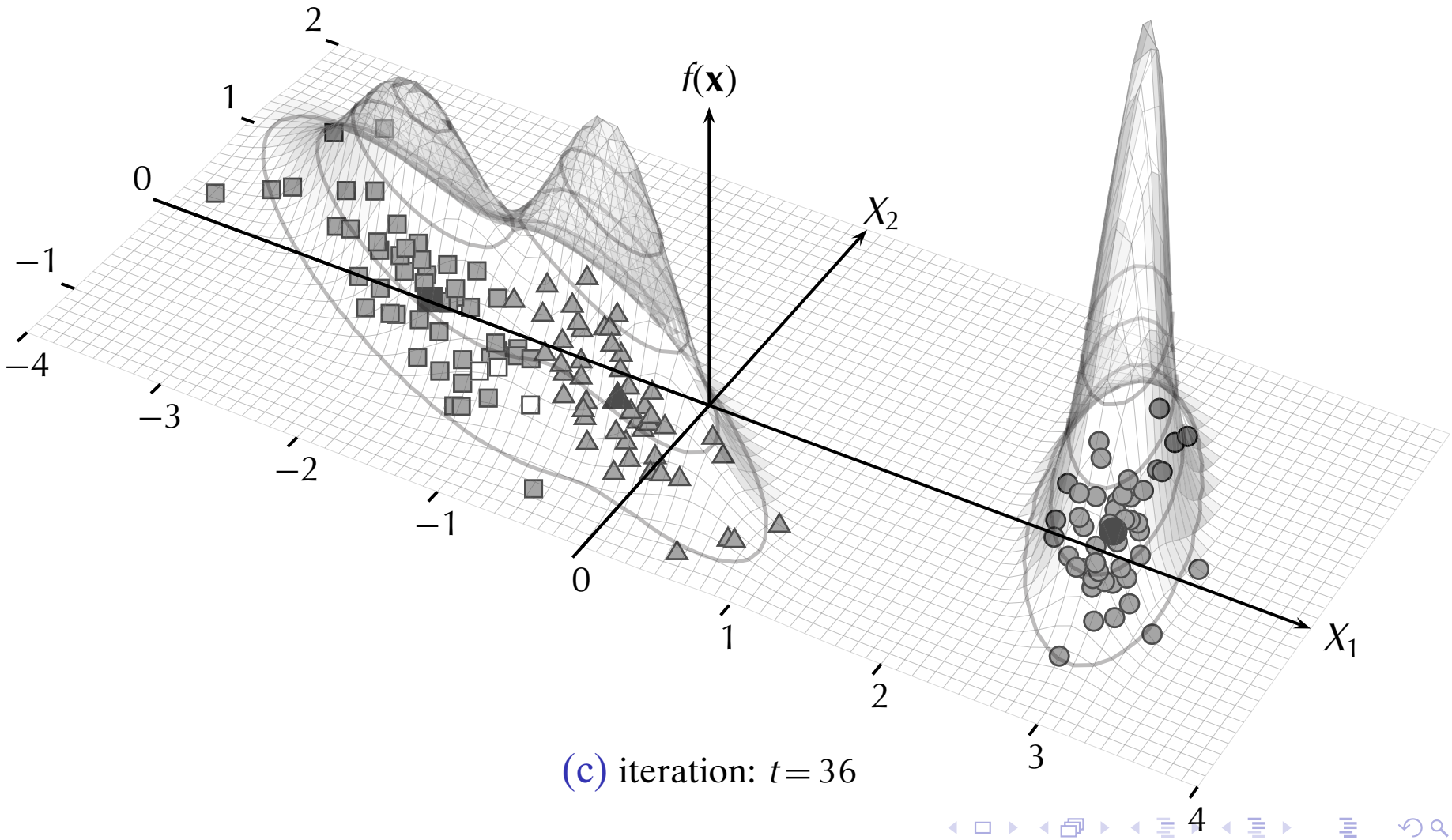
Mixture of $k = 3$ Gaussians



(b) Iteration: $t = 1$

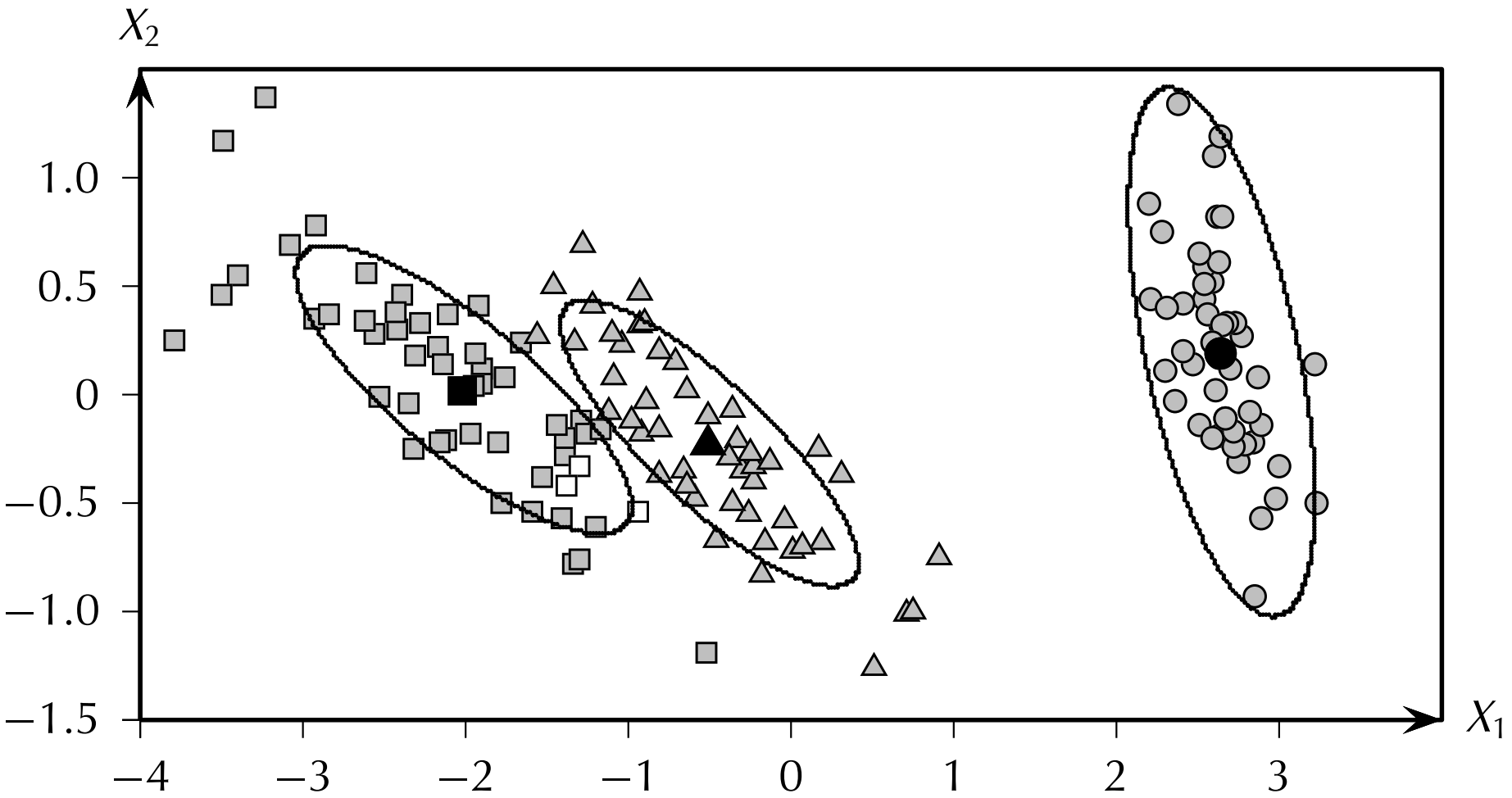
EM Clustering in 2D

Mixture of $k = 3$ Gaussians



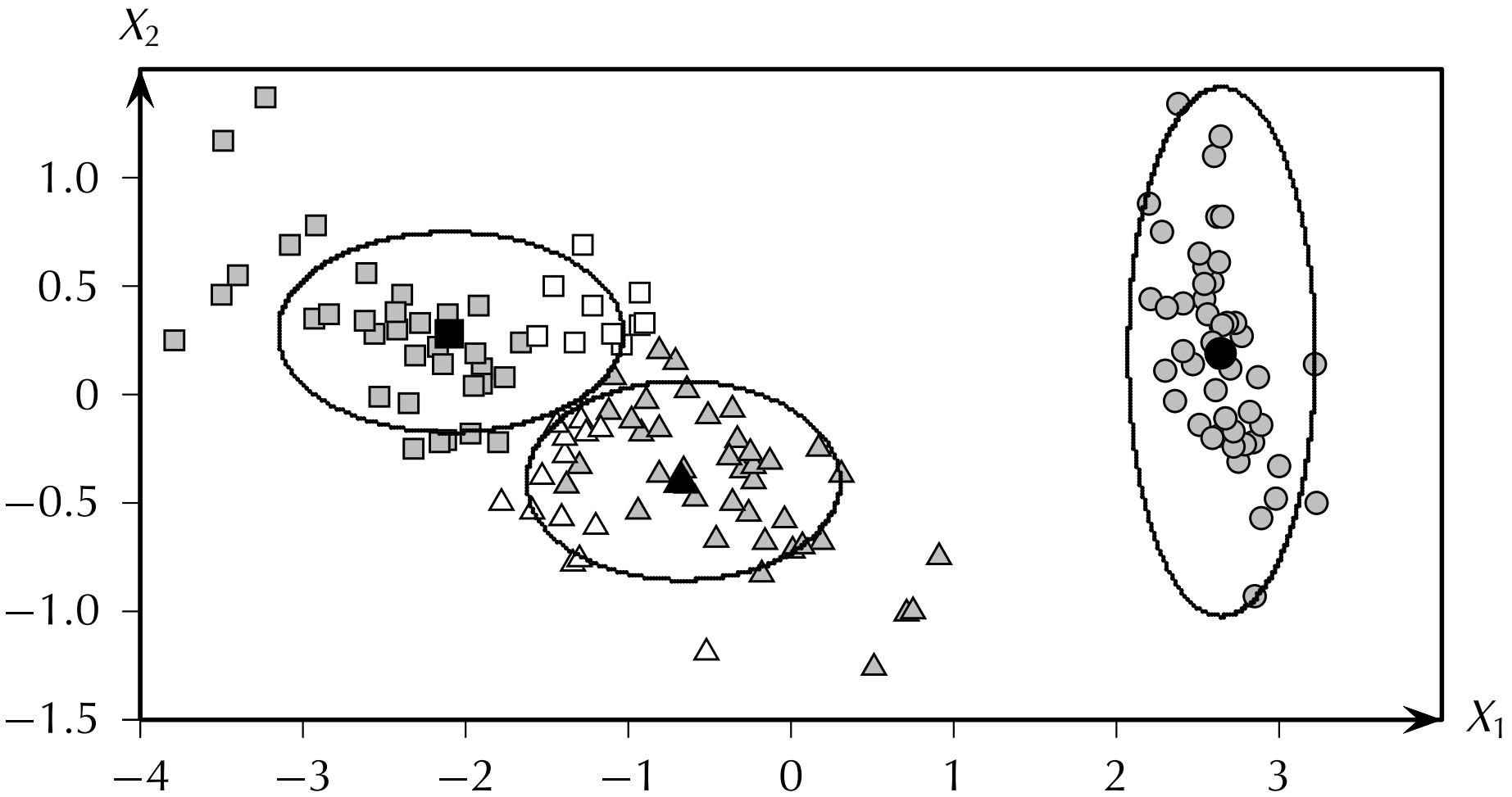
(c) iteration: $t = 36$

Iris Principal Components Data: Full Covariance Matrix



(a) Full covariance matrix ($t = 36$)

Iris Principal Components Data: Diagonal Covariance Matrix



(b) Diagonal covariance matrix ($t = 29$)

- Expectation-Maximization
- **DenClue**
- Naive Bayes

Kernel Density Estimation

There is a close connection between density-based clustering and density estimation. The goal of density estimation is to determine the unknown probability density function by finding the dense regions of points, which can in turn be used for clustering.

Kernel density estimation is a nonparametric technique that does not assume any fixed probability model of the clusters. Instead, it tries to directly infer the underlying probability density at each point in the dataset.

Univariate Density Estimation

Assume that X is a continuous random variable, and let $x_1, x_2, \ldots, x_n$ be a random sample. We directly estimate the cumulative distribution function from the data by counting how many points are less than or equal to x:

$$\hat{F}(x) = \frac{1}{n} \sum_{i=1}^{n} I(x_i \leq x)$$

where I is an indicator function.

We estimate the density function by taking the derivative of $\hat{F}(x)$

$$\hat{f}(x) = \frac{\hat{F}\left(x + \frac{h}{2}\right) - \hat{F}\left(x - \frac{h}{2}\right)}{h} = \frac{k/n}{h} = \frac{k}{nh}$$

where k is the number of points that lie in the window of width h centered at x. The density estimate is the ratio of the fraction of the points in the window (k/n) to the volume of the window (h).

subsubsectionKernel Estimator Kernel density estimation relies on a *kernel function* K that is non-negative, symmetric, and integrates to 1, that is, $K(x) \geq 0$, $K(-x) = K(x)$ for all values x, and $\int K(x)dx = 1$.

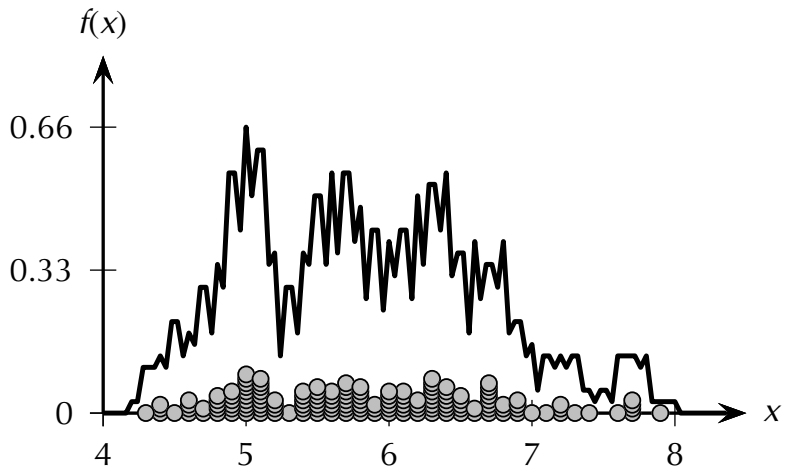
Discrete Kernel Define the **discrete kernel** function K, that computes the number of points in a window of width h

$$K(z) = \begin{cases} 1 & \text{If } |z| \leq \frac{1}{2} \\ 0 & \text{Otherwise} \end{cases}$$

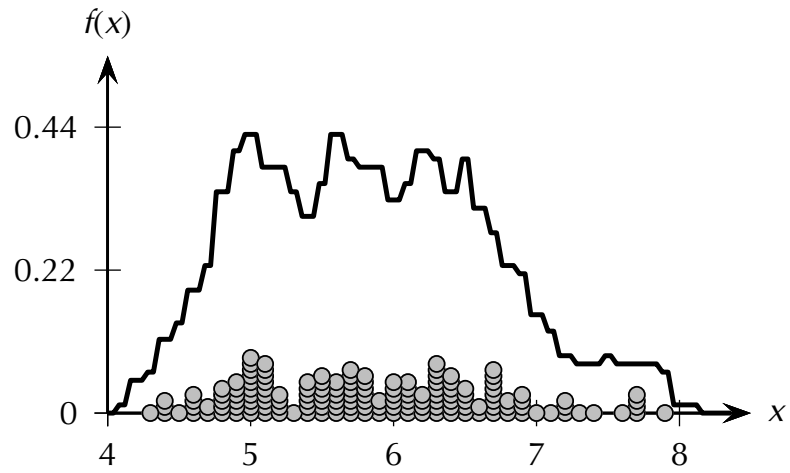
The density estimate $\hat{f}(x)$ can be rewritten in terms of the kernel function as follows:

$$\hat{f}(x) = \frac{1}{nh} \sum_{i=1}^{n} K\left(\frac{x - x_i}{h}\right)$$

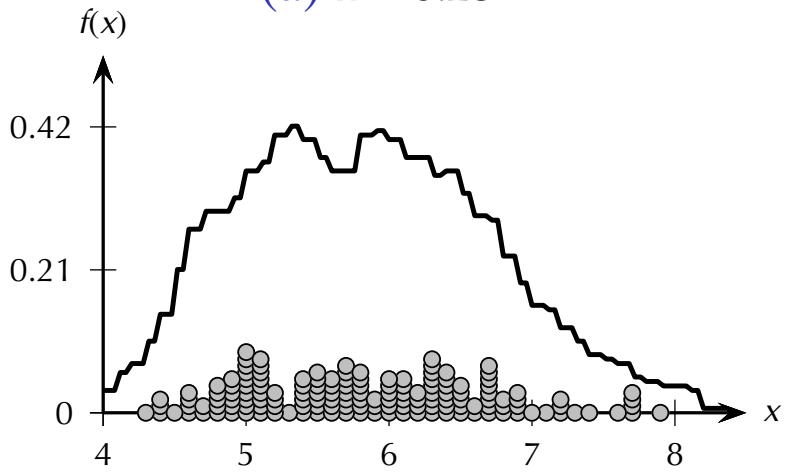
Kernel Density Estimation: Discrete Kernel (Iris 1D)



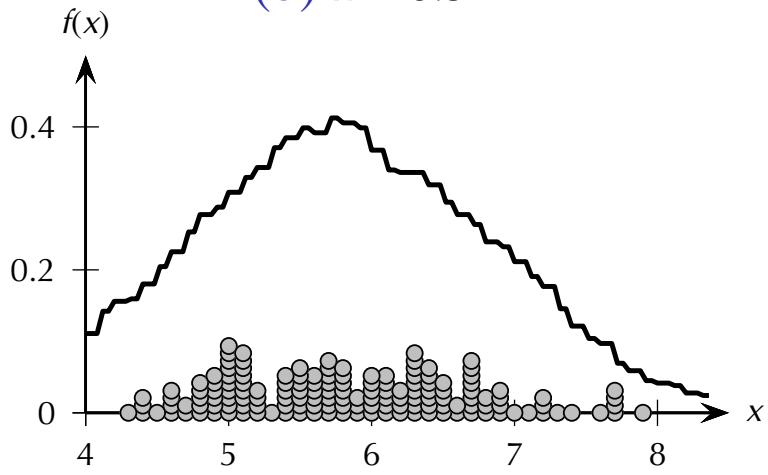
(a) $h = 0.25$



(b) $h = 0.5$



(c) $h = 1.0$



(d) $h = 2.0$

The discrete kernel yields a non-smooth (or jagged) density function.

kernel Density Estimation: Gaussian Kernel

The width h is a parameter that denotes the spread or smoothness of the density estimate. The discrete kernel function has an abrupt influence.

Define a more smooth transition of influence via a Gaussian kernel:

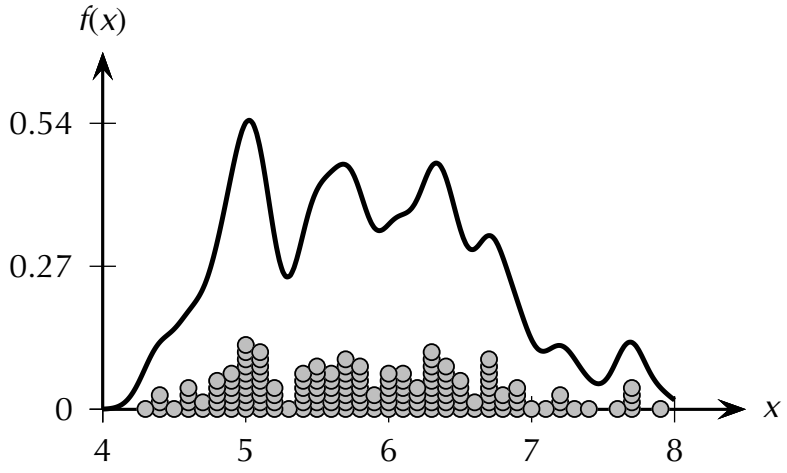
$$K(z) = \frac{1}{\sqrt{2\pi}} \exp\left\{-\frac{z^2}{2}\right\}$$

Thus, we have

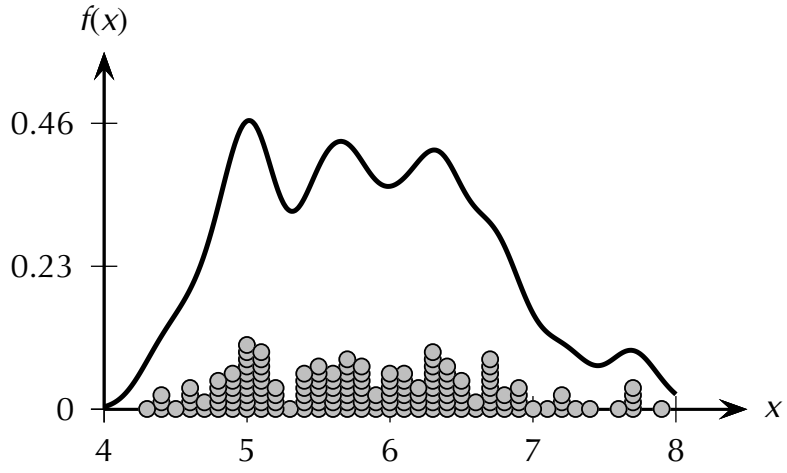
$$K\left(\frac{x - x_i}{h}\right) = \frac{1}{\sqrt{2\pi}} \exp\left\{-\frac{(x - x_i)^2}{2h^2}\right\}$$

Here x, which is at the center of the window, plays the role of the mean, and h acts as the standard deviation.

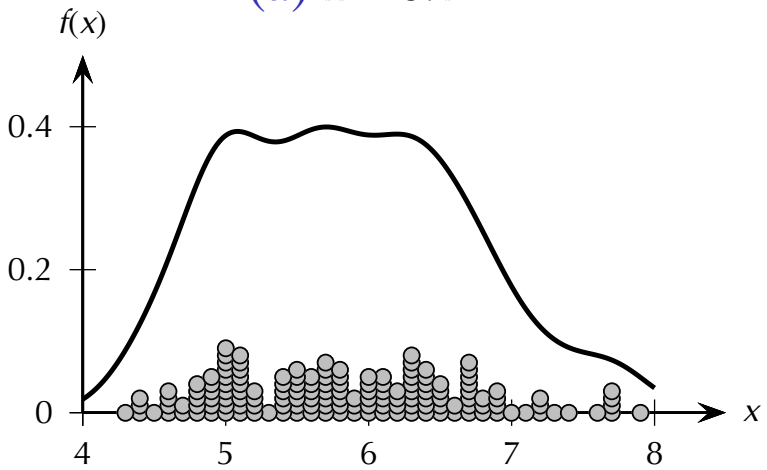
Kernel Density Estimation: Gaussian Kernel (Iris 1D)



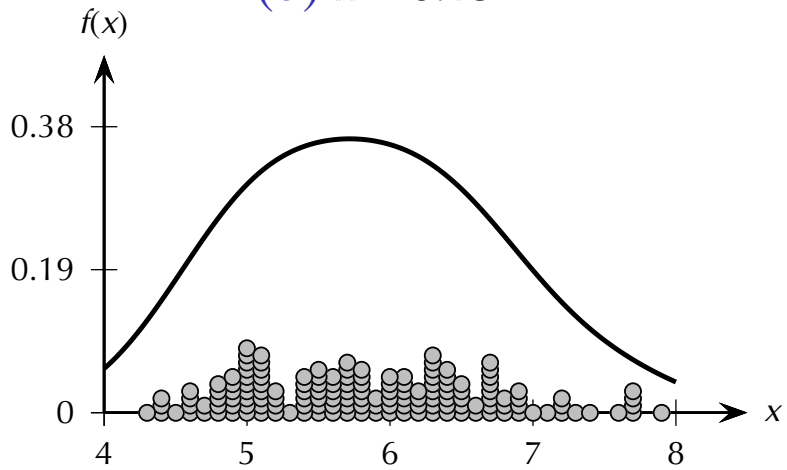
(a) $h = 0.1$



(b) $h = 0.15$



(c) $h = 0.25$



(d) $h = 0.5$

When h is small the density function has many local maxima. A large h results in a unimodal distribution.

Multivariate Density Estimation

To estimate the probability density at a d-dimensional point $\mathbf{x} = (x_1, x_2, \ldots, x_d)^T$, we define the d-dimensional "window" as a hypercube in d dimensions, that is, a hypercube centered at $\mathbf{x}$ with edge length h. The volume of such a d-dimensional hypercube is given as

$$\text{vol}(H_d(h)) = h^d$$

The density is estimated as the fraction of the point weight lying within the d-dimensional window centered at $\mathbf{x}$, divided by the volume of the hypercube:

$$\hat{f}(\mathbf{x}) = \frac{1}{nh^d} \sum_{i=1}^{n} K\left(\frac{\mathbf{x} - \mathbf{x}_i}{h}\right)$$

where the multivariate kernel function K satisfies the condition $\int K(\mathbf{z})d\mathbf{z} = 1$.

Multivariate Density Estimation: Discrete and Gaussian Kernel

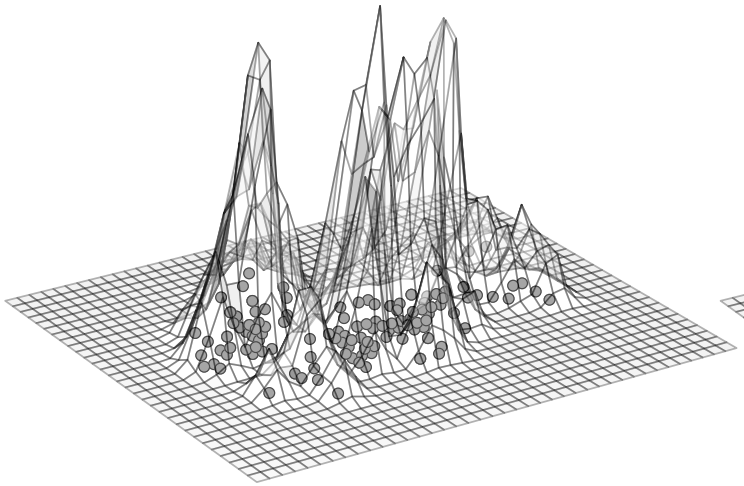
Discrete Kernel: For any d-dimensional vector $\mathbf{z} = (z_1, z_2, \ldots, z_d)^T$, the discrete kernel function in d-dimensions is given as

$$K(\mathbf{z}) = \begin{cases} 1 & \text{If } |z_j| \leq \frac{1}{2}, \text{ for all dimensions } j = 1, \ldots, d \\ 0 & \text{Otherwise} \end{cases}$$

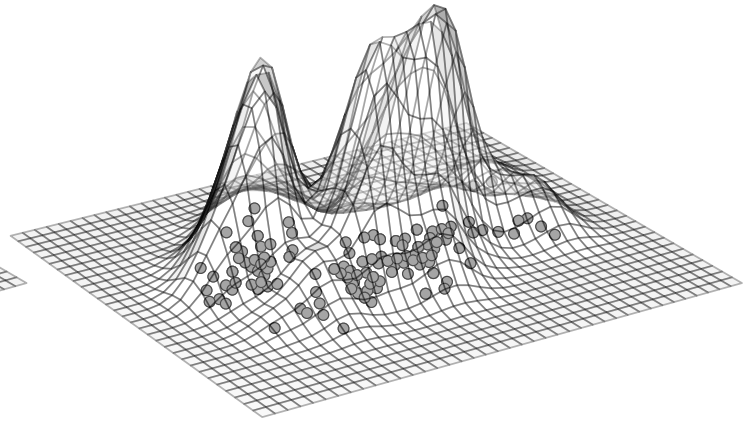
Gaussian Kernel: The d-dimensional Gaussian kernel is given as

$$K(\mathbf{z}) = \frac{1}{(2\pi)^{d/2}} \exp\left\{-\frac{\mathbf{z}^T\mathbf{z}}{2}\right\}$$

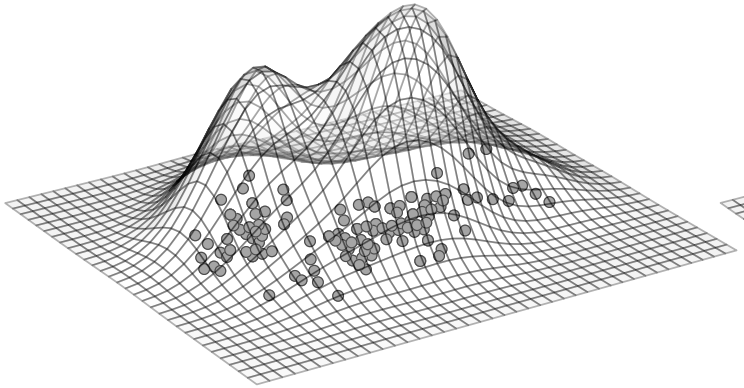
Density Estimation: Iris 2D Data (Gaussian Kernel)



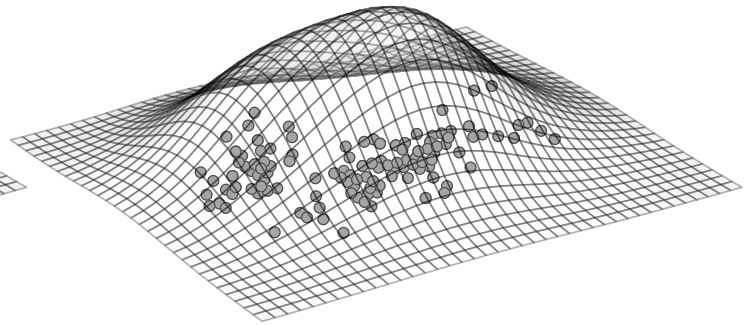
(a) $h = 0.1$



(b) $h = 0.2$



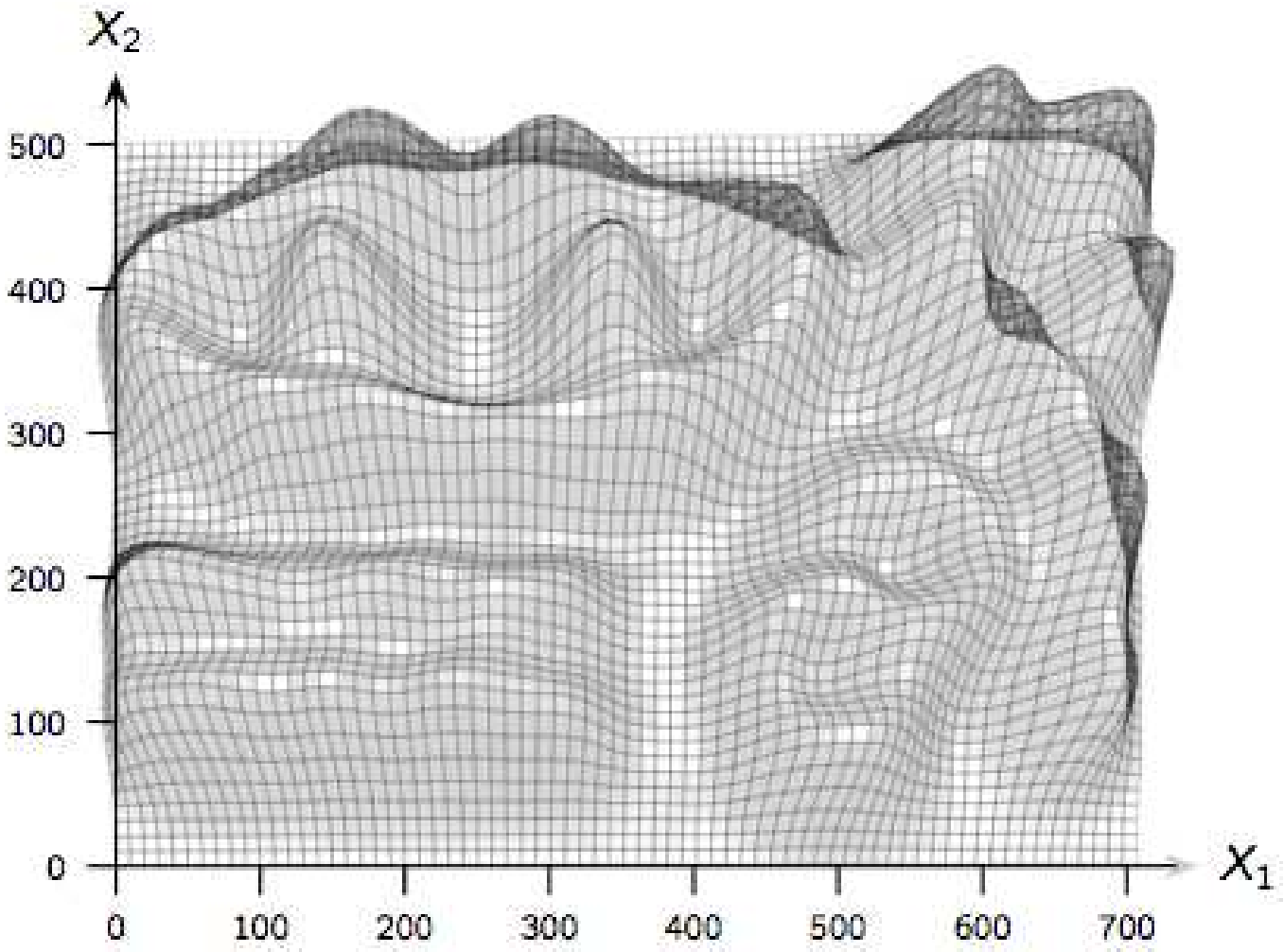
(c) $h = 0.35$



(d) $h = 0.6$

Density Estimation: Density-based Dataset

Gaussian kernel, $h = 20$



Nearest Neighbor Density Estimation

In kernel density estimation we implicitly fixed the volume by fixing the width h, and we used the kernel function to find out the number or weight of points that lie inside the fixed volume region.

An alternative approach to density estimation is to fix k, the number of points required to estimate the density, and allow the volume of the enclosing region to vary to accommodate those k points. This approach is called the k nearest neighbors (KNN) approach to density estimation.

Given k, the number of neighbors, we estimate the density at $\mathbf{x}$ as follows:

$$\hat{f}(\mathbf{x}) = \frac{k}{n \text{vol}(S_d(h_\mathbf{x}))}$$

where $h_\mathbf{x}$ is the distance from $\mathbf{x}$ to its kth nearest neighbor, and $\text{vol}(S_d(h_\mathbf{x}))$ is the volume of the d-dimensional hypersphere $S_d(h_\mathbf{x})$ centered at $\mathbf{x}$, with radius $h_\mathbf{x}$.

DENCLUE Density-based Clustering: Attractor and Gradient

A point $\mathbf{x}^*$ is called a *density attractor* if it is a local maxima of the probability density function f.

The density gradient at a point $\mathbf{x}$ is the multivariate derivative of the probability density estimate

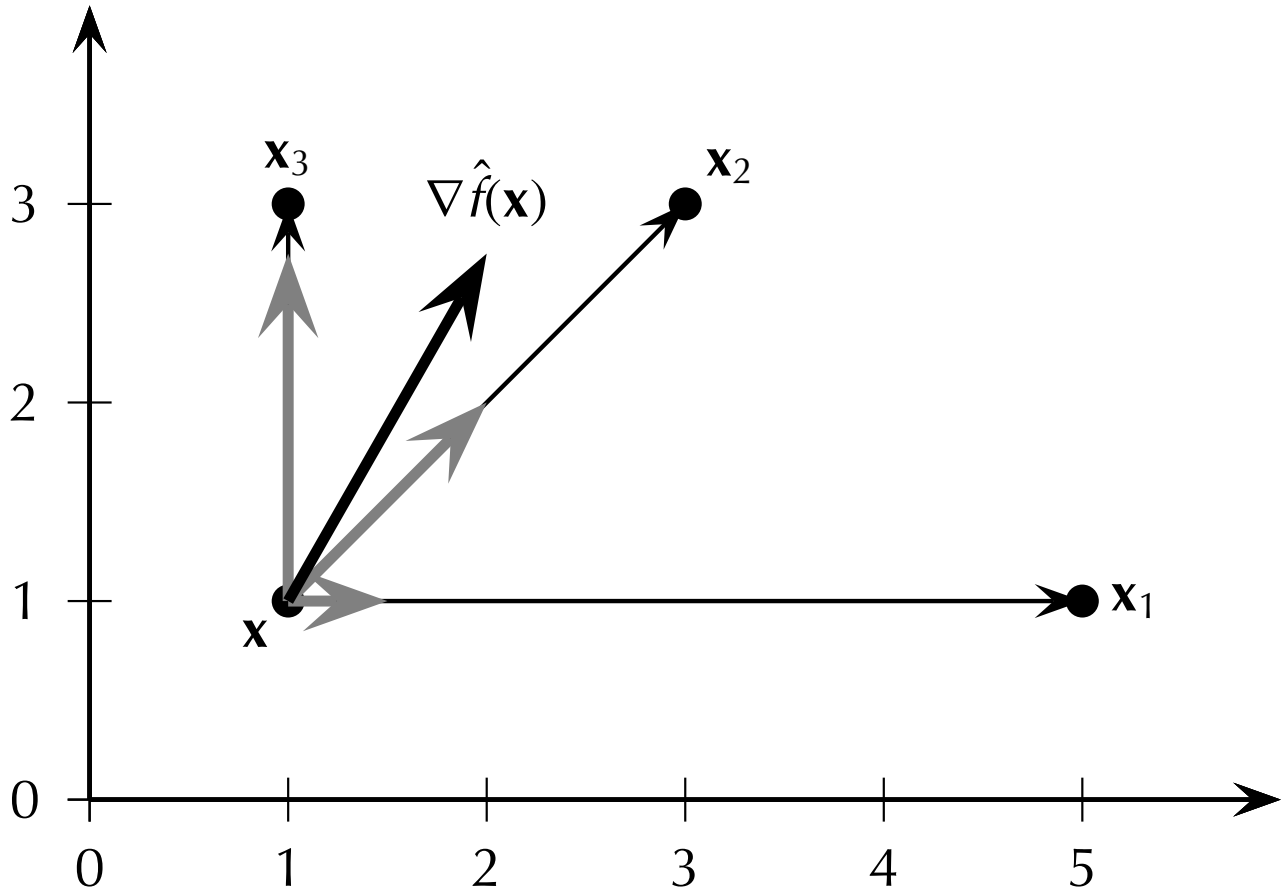
$$\nabla \hat{f}(\mathbf{x}) = \frac{\partial}{\partial \mathbf{x}} \hat{f}(\mathbf{x}) = \frac{1}{nh^d} \sum_{i=1}^{n} \frac{\partial}{\partial \mathbf{x}} K\left(\frac{\mathbf{x} - \mathbf{x}_i}{h}\right)$$

For the Gaussian kernel the gradient at a point $\mathbf{x}$ is given as

$$\nabla \hat{f}(\mathbf{x}) = \frac{1}{nh^{d+2}} \sum_{i=1}^{n} K\left(\frac{\mathbf{x} - \mathbf{x}_i}{h}\right) \cdot (\mathbf{x}_i - \mathbf{x})$$

This equation can be thought of as having two parts for each point: a vector $(\mathbf{x}_i - \mathbf{x})$ and a scalar *influence* value $K(\frac{\mathbf{x}-\mathbf{x}_i}{h})$.

The Gradient Vector



DENCLUE: Density Attractor

We say that $\mathbf{x}^*$ is a *density attractor* for $\mathbf{x}$, or alternatively that $\mathbf{x}$ is *density attracted* to $\mathbf{x}^*$, if a hill climbing process started at $\mathbf{x}$ converges to $\mathbf{x}^*$.

That is, there exists a sequence of points $\mathbf{x} = \mathbf{x}_0 \rightarrow \mathbf{x}_1 \rightarrow \ldots \rightarrow \mathbf{x}_m$, starting from $\mathbf{x}$ and ending at $\mathbf{x}_m$, such that $\|\mathbf{x}_m - \mathbf{x}^*\| \leq \epsilon$, that is, $\mathbf{x}_m$ converges to the attractor $\mathbf{x}^*$.

Setting the gradient to the zero vector leads to the following *mean-shift* update rule:

$$\mathbf{x}_{t+1} = \frac{\sum_{i=1}^{n} K\left(\frac{\mathbf{x}_t - \mathbf{x}_i}{h}\right) \mathbf{x}_i}{\sum_{i=1}^{n} K\left(\frac{\mathbf{x}_t - \mathbf{x}_i}{h}\right)}$$

where t denotes the current iteration and $\mathbf{x}_{t+1}$ is the updated value for the current vector $\mathbf{x}_t$.

DENCLUE: Density-based Cluster

A cluster $C \subseteq \mathbf{D}$, is called a *center-defined cluster* if all the points $\mathbf{x} \in C$ are density attracted to a unique density attractor $\mathbf{x}^*$, such that $\hat{f}(\mathbf{x}^*) \geq \xi$, where ξ is a user-defined minimum density threshold.

An arbitrary-shaped cluster $C \subseteq \mathbf{D}$ is called a *density-based cluster* if there exists a set of density attractors $\mathbf{x}_1^*, \mathbf{x}_2^*, \ldots, \mathbf{x}_m^*$, such that

1. Each point $\mathbf{x} \in C$ is attracted to some attractor $\mathbf{x}_i^*$.
2. Each density attractor has density above ξ.
3. Any two density attractors $\mathbf{x}_i^*$ and $\mathbf{x}_j^*$ are *density reachable*, that is, there exists a path from $\mathbf{x}_i^*$ to $\mathbf{x}_j^*$, such that for all points $\mathbf{y}$ on the path, $\hat{f}(\mathbf{y}) \geq \xi$.

The DENCLUE Algorithm

DENCLUE $(\mathbf{D}, h, \xi, \epsilon)$:

$\mathcal{A} \leftarrow \emptyset$

foreach $\mathbf{x} \in \mathbf{D}$ **do** // find density attractors

  $\mathbf{x}^* \leftarrow \text{FINDATTRACTOR}(\mathbf{x}, \mathbf{D}, h, \epsilon)$

  if $\hat{f}(\mathbf{x}^*) \geq \xi$ **then**

    $\mathcal{A} \leftarrow \mathcal{A} \cup \{\mathbf{x}^*\}$

    $R(\mathbf{x}^*) \leftarrow R(\mathbf{x}^*) \cup \{\mathbf{x}\}$

$\mathcal{C} \leftarrow \{\text{maximal } C \subseteq \mathcal{A} \mid \forall \mathbf{x}_i^*, \mathbf{x}_j^* \in C, \mathbf{x}_i^* \text{ and } \mathbf{x}_j^* \text{ are density reachable}\}$

foreach $C \in \mathcal{C}$ **do** // density-based clusters

  foreach $\mathbf{x}^* \in C$ **do** $C \leftarrow C \cup R(\mathbf{x}^*)$

return $\mathcal{C}$

The DENCLUE Algorithm: Find Attractor

FINDATTRACTOR $(\mathbf{x}, \mathbf{D}, h, \epsilon)$:

$t \leftarrow 0$

$\mathbf{x}_t \leftarrow \mathbf{x}$

repeat

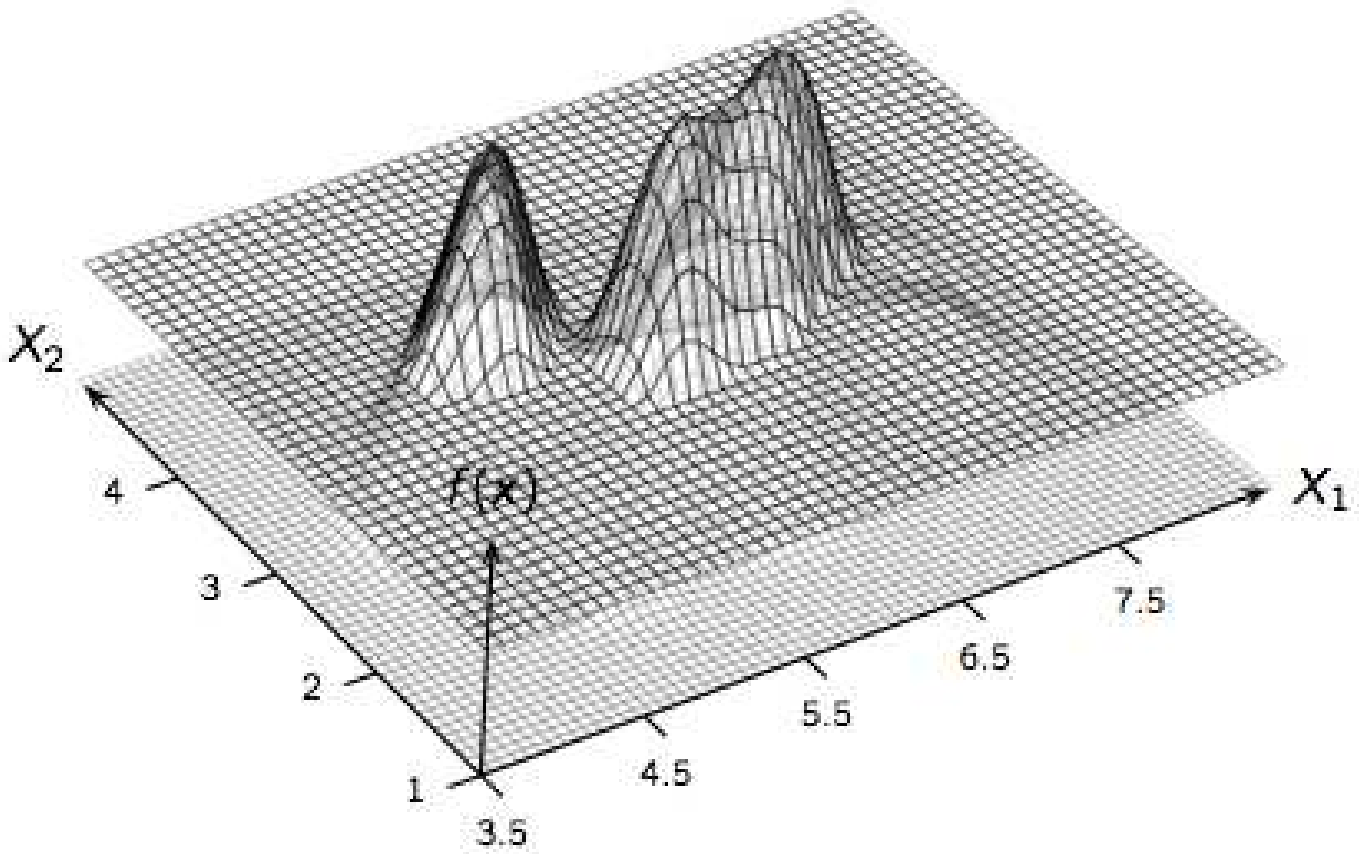
$$\mathbf{x}_{t+1} \leftarrow \frac{\sum_{i=1}^{n} K\left(\frac{\mathbf{x}_t - \mathbf{x}_i}{h}\right) \cdot \mathbf{x}_t}{\sum_{i=1}^{n} K\left(\frac{\mathbf{x}_t - \mathbf{x}_i}{h}\right)}$$

$t \leftarrow t + 1$

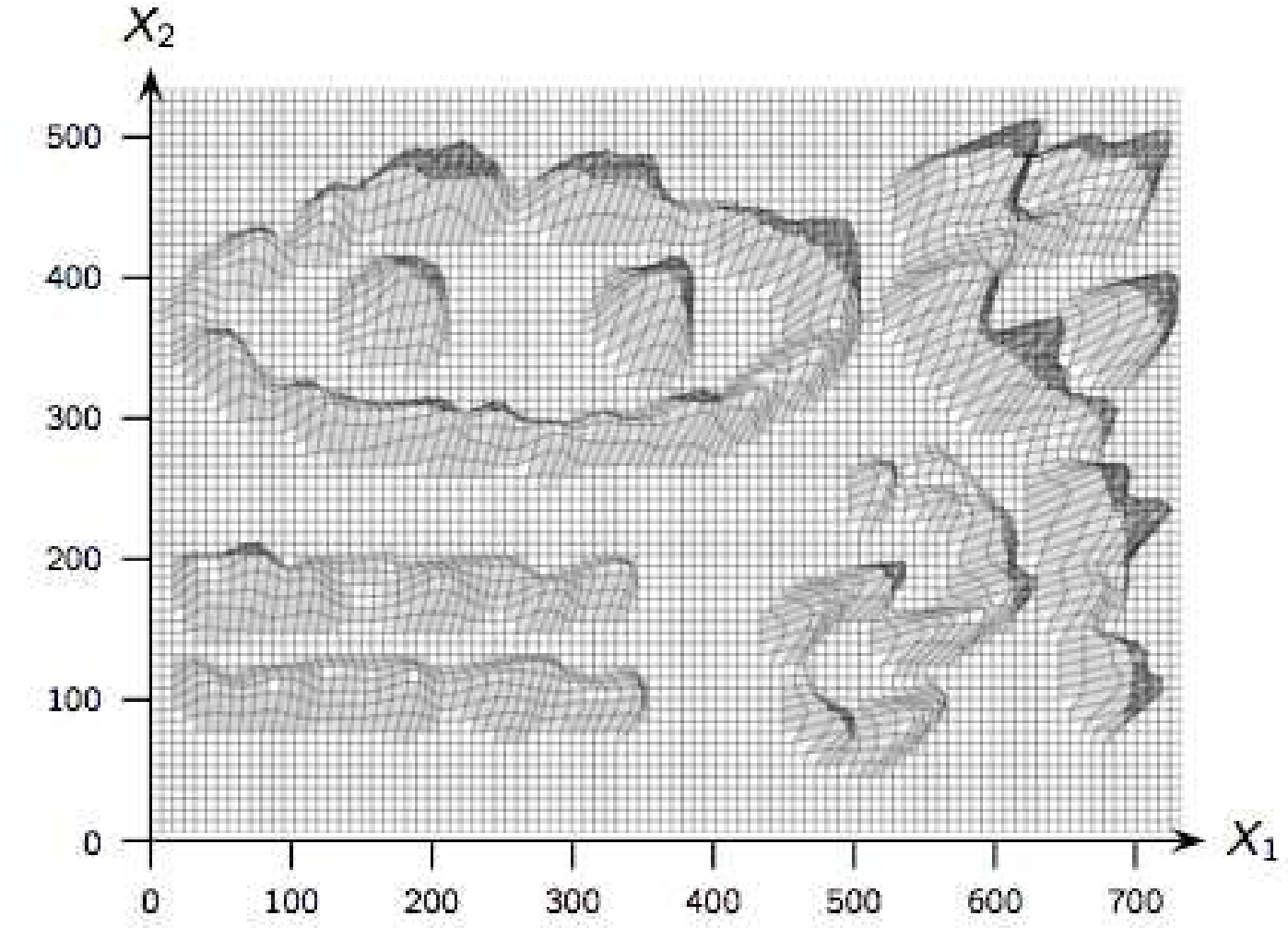
until $\|\mathbf{x}_t - \mathbf{x}_{t-1}\| \leq \epsilon$

return $\mathbf{x}_t$

DENCLUE: Iris 2D Data



DENCLUE: Density-based Dataset



- Expectation-Maximization
- DenClue
- **Naive Bayes**

Bayes Classifier

Let the training dataset $\mathbf{D}$ consist of n points $\mathbf{x}_i$ in a d-dimensional space, and let y_i denote the class for each point, with $y_i \in \{c_1, c_2, \ldots, c_k\}$.

The Bayes classifier estimates the posterior probability $P(c_i|\mathbf{x})$ for each class c_i, and chooses the class that has the largest probability. The predicted class for $\mathbf{x}$ is given as

$$\hat{y} = \arg\max_{c_i}\{P(c_i|\mathbf{x})\}$$

According to the Bayes theorem, we have

$$P(c_i|\mathbf{x}) = \frac{P(\mathbf{x}|c_i) \cdot P(c_i)}{P(\mathbf{x})}$$

Because $P(\mathbf{x})$ is fixed for a given point, Bayes rule can be rewritten as

$$\hat{y} = \arg\max_{c_i}\{P(c_i|\mathbf{x})\} = \arg\max_{c_i}\left\{\frac{P(\mathbf{x}|c_i)P(c_i)}{P(\mathbf{x})}\right\} = \arg\max_{c_i}\left\{P(\mathbf{x}|c_i)P(c_i)\right\}$$

Estimating the Prior Probability: $P(c_i)$

Let $\mathbf{D}_i$ denote the subset of points in $\mathbf{D}$ that are labeled with class c_i:

$$\mathbf{D}_i = \{\mathbf{x}_j \in \mathbf{D} \mid \mathbf{x}_j \text{ has class } y_j = c_i\}$$

Let the size of the dataset $\mathbf{D}$ be given as $|\mathbf{D}| = n$, and let the size of each class-specific subset $\mathbf{D}_i$ be given as $|\mathbf{D}_i| = n_i$.

The prior probability for class c_i can be estimated as follows:

$$\hat{P}(c_i) = \frac{n_i}{n}$$

Estimating the Likelihood: Numeric Attributes, Parametric Approach

To estimate the likelihood $P(\mathbf{x}|c_i)$, we have to estimate the joint probability of $\mathbf{x}$ across all the d dimensions, i.e., we have to estimate $P\big(\mathbf{x} = (x_1, x_2, \ldots, x_d)|c_i\big)$.

In the parametric approach we assume that each class c_i is normally distributed, and we use the estimated mean $\hat{\boldsymbol{\mu}}_i$ and covariance matrix $\widehat{\boldsymbol{\Sigma}}_i$ to compute the probability density at $\mathbf{x}$

$$\hat{f}_i(\mathbf{x}) = \hat{f}(\mathbf{x}|\hat{\boldsymbol{\mu}}_i, \widehat{\boldsymbol{\Sigma}}_i) = \frac{1}{(\sqrt{2\pi})^d \sqrt{|\widehat{\boldsymbol{\Sigma}}_i|}} \exp\left\{-\frac{(\mathbf{x} - \hat{\boldsymbol{\mu}}_i)^T \widehat{\boldsymbol{\Sigma}}_i^{-1} (\mathbf{x} - \hat{\boldsymbol{\mu}}_i)}{2}\right\}$$

The posterior probability is then given as

$$P(c_i|\mathbf{x}) = \frac{\hat{f}_i(\mathbf{x}) P(c_i)}{\sum_{j=1}^{k} \hat{f}_j(\mathbf{x}) P(c_j)}$$

The predicted class for $\mathbf{x}$ is:

$$\hat{y} = \arg\max_{c_i}\left\{\hat{f}_i(\mathbf{x}) P(c_i)\right\}$$

Bayes Classifier Algorithm

BayesClassifier ($\mathbf{D} = \{(\mathbf{x}_j, y_j)\}_{j=1}^{n}$):

for $i = 1, \ldots, k$ **do**
- $\mathbf{D}_i \leftarrow \{\mathbf{x}_j \mid y_j = c_i, j = 1, \ldots, n\}$ // class-specific subsets
- $n_i \leftarrow |\mathbf{D}_i|$ // cardinality
- $\hat{P}(c_i) \leftarrow n_i/n$ // prior probability
- $\hat{\boldsymbol{\mu}}_i \leftarrow \frac{1}{n_i} \sum_{\mathbf{x}_j \in \mathbf{D}_i} \mathbf{x}_j$ // mean
- $\mathbf{Z}_i \leftarrow \mathbf{D}_i - \mathbf{1}_{n_i} \hat{\boldsymbol{\mu}}_i^T$ // centered data
- $\widehat{\boldsymbol{\Sigma}}_i \leftarrow \frac{1}{n_i} \mathbf{Z}_i^T \mathbf{Z}_i$ // covariance matrix

return $\hat{P}(c_i), \hat{\boldsymbol{\mu}}_i, \widehat{\boldsymbol{\Sigma}}_i$ *for all* $i = 1, \ldots, k$

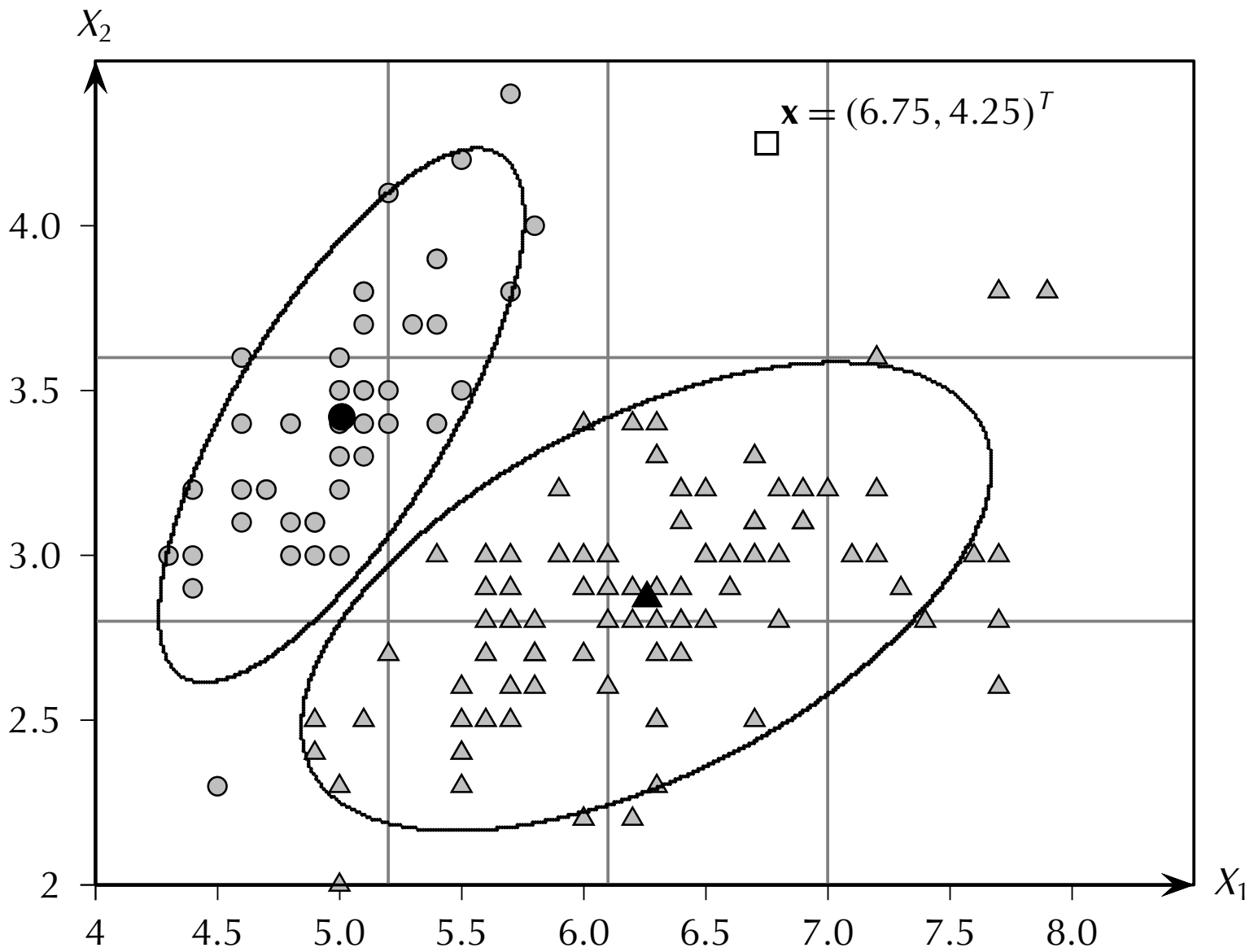
Testing (**x** and $\hat{P}(c_i)$, $\hat{\boldsymbol{\mu}}_i$, $\widehat{\boldsymbol{\Sigma}}_i$, **for all** $i \in [1, k]$):

$\hat{y} \leftarrow \arg\max_{c_i} \left\{ f(\mathbf{x} | \hat{\boldsymbol{\mu}}_i, \widehat{\boldsymbol{\Sigma}}_i) \cdot P(c_i) \right\}$

return $\hat{y}$

Bayes Classifier: Iris Data

X_1:sepal length versus X_2:sepal width



Bayes Classifier: Categorical Attributes

Let X_j be a categorical attribute over the domain $dom(X_j) = \{a_{j1}, a_{j2}, \ldots, a_{jm_j}\}$. Each categorical attribute X_j is modeled as an m_j-dimensional multivariate Bernoulli random variable $\mathbf{X}_j$ that takes on m_j distinct vector values $\mathbf{e}_{j1}, \mathbf{e}_{j2}, \ldots, \mathbf{e}_{jm_j}$, where $\mathbf{e}_{jr}$ is the rth standard basis vector in $\mathbb{R}^{m_j}$ and corresponds to the rth value or symbol $a_{jr} \in dom(X_j)$.

The entire d-dimensional dataset is modeled as the vector random variable $\mathbf{X} = (\mathbf{X}_1, \mathbf{X}_2, \ldots, \mathbf{X}_d)^T$. Let $d' = \sum_{j=1}^{d} m_j$; a categorical point $\mathbf{x} = (x_1, x_2, \ldots, x_d)^T$ is therefore represented as the d'-dimensional binary vector

$$\mathbf{v} = \begin{pmatrix} \mathbf{v}_1 \\ \vdots \\ \mathbf{v}_d \end{pmatrix} = \begin{pmatrix} \mathbf{e}_{1r_1} \\ \vdots \\ \mathbf{e}_{dr_d} \end{pmatrix}$$

where $\mathbf{v}_j = \mathbf{e}_{jr_j}$ provided $x_j = a_{jr_j}$ is the r_jth value in the domain of X_j.

Bayes Classifier: Categorical Attributes

The probability of the categorical point $\mathbf{x}$ is obtained from the joint probability mass function (PMF) for the vector random variable $\mathbf{X}$:

$$P(\mathbf{x}|c_i) = f(\mathbf{v}|c_i) = f\big(\mathbf{X}_1 = \mathbf{e}_{1r_1}, \ldots, \mathbf{X}_d = \mathbf{e}_{dr_d} | \; c_i\big)$$

The joint PMF can be estimated directly from the data $\mathbf{D}_i$ for each class c_i as follows:

$$\hat{f}(\mathbf{v}|c_i) = \frac{n_i(\mathbf{v})}{n_i}$$

where $n_i(\mathbf{v})$ is the number of times the value $\mathbf{v}$ occurs in class c_i.
However, to avoid zero probabilities we add a *pseudo-count* of 1 for each value

$$\hat{f}(\mathbf{v}|c_i) = \frac{n_i(\mathbf{v}) + 1}{n_i + \prod_{j=1}^{d} m_j}$$

Discretized Iris Data: `sepal length` and `sepal width`

Bins	Domain
$[4.3, 5.2]$	Very Short (a_{11})
$(5.2, 6.1]$	Short (a_{12})
$(6.1, 7.0]$	Long (a_{13})
$(7.0, 7.9]$	Very Long (a_{14})

(a) Discretized `sepal length`

Bins	Domain
$[2.0, 2.8]$	Short (a_{21})
$(2.8, 3.6]$	Medium (a_{22})
$(3.6, 4.4]$	Long (a_{23})

(b) Discretized `sepal width`

Class-specific Empirical Joint Probability Mass Function

	Class: c_1	X_2			$\hat{f}_{X_1}$
		Short ($\mathbf{e}_{21}$)	Medium ($\mathbf{e}_{22}$)	Long ($\mathbf{e}_{23}$)	
X_1	Very Short ($\mathbf{e}_{11}$)	1/50	33/50	5/50	39/50
	Short ($\mathbf{e}_{12}$)	0	3/50	8/50	13/50
	Long ($\mathbf{e}_{13}$)	0	0	0	0
	Very Long ($\mathbf{e}_{14}$)	0	0	0	0
	$\hat{f}_{X_2}$	1/50	36/50	13/50	

	Class: c_2	X_2			$\hat{f}_{X_1}$
		Short ($\mathbf{e}_{21}$)	Medium ($\mathbf{e}_{22}$)	Long ($\mathbf{e}_{23}$)	
X_1	Very Short ($\mathbf{e}_{11}$)	6/100	0	0	6/100
	Short ($\mathbf{e}_{12}$)	24/100	15/100	0	39/100
	Long ($\mathbf{e}_{13}$)	13/100	30/100	0	43/100
	Very Long ($\mathbf{e}_{14}$)	3/100	7/100	2/100	12/100
	$\hat{f}_{X_2}$	46/100	52/100	2/100	

Iris Data: Test Case

Consider a test point $\mathbf{x} = (5.3, 3.0)^T$ corresponding to the categorical point (`Short, Medium`), which is represented as $\mathbf{v} = \begin{pmatrix} \mathbf{e}_{12}^T & \mathbf{e}_{22}^T \end{pmatrix}^T$.

The prior probabilities of the classes are $\hat{P}(c_1) = 0.33$ and $\hat{P}(c_2) = 0.67$. The likelihood and posterior probability for each class is given as

$$\hat{P}(\mathbf{x}|c_1) = \hat{f}(\mathbf{v}|c_1) = 3/50 = 0.06$$
$$\hat{P}(\mathbf{x}|c_2) = \hat{f}(\mathbf{v}|c_2) = 15/100 = 0.15$$
$$\hat{P}(c_1|\mathbf{x}) \propto 0.06 \times 0.33 = 0.0198$$
$$\hat{P}(c_2|\mathbf{x}) \propto 0.15 \times 0.67 = 0.1005$$

In this case the predicted class is $\hat{y} = c_2$.

Iris Data: Test Case with Pseudo-counts

The test point $\mathbf{x} = (6.75, 4.25)^T$ corresponds to the categorical point (`Long`, `Long`), and it is represented as $\mathbf{v} = \begin{pmatrix} \mathbf{e}_{13}^T & \mathbf{e}_{23}^T \end{pmatrix}^T$.

Unfortunately the probability mass at $\mathbf{v}$ is zero for both classes. We adjust the PMF via pseudo-counts noting that the number of possible values are $m_1 \times m_2 = 4 \times 3 = 12$.

The likelihood and prior probability can then be computed as

$$\hat{P}(\mathbf{x}|c_1) = \hat{f}(\mathbf{v}|c_1) = \frac{0+1}{50+12} = 1.61 \times 10^{-2}$$

$$\hat{P}(\mathbf{x}|c_2) = \hat{f}(\mathbf{v}|c_2) = \frac{0+1}{100+12} = 8.93 \times 10^{-3}$$

$$\hat{P}(c_1|\mathbf{x}) \propto (1.61 \times 10^{-2}) \times 0.33 = 5.32 \times 10^{-3}$$

$$\hat{P}(c_2|\mathbf{x}) \propto (8.93 \times 10^{-3}) \times 0.67 = 5.98 \times 10^{-3}$$

Thus, the predicted class is $\hat{y} = c_2$.

Bayes Classifier: Challenges

The main problem with the Bayes classifier is the lack of enough data to reliably estimate the joint probability density or mass function, especially for high-dimensional data.

For numeric attributes we have to estimate $O(d^2)$ covariances, and as the dimensionality increases, this requires us to estimate too many parameters.

For categorical attributes we have to estimate the joint probability for all the possible values of $\mathbf{v}$, given as $\prod_j |\text{dom}(X_j)|$. Even if each categorical attribute has only two values, we would need to estimate the probability for 2^d values. However, because there can be at most n distinct values for $\mathbf{v}$, most of the counts will be zero.

Naive Bayes classifier addresses these concerns.

Naive Bayes Classifier: Numeric Attributes

The naive Bayes approach makes the simple assumption that all the attributes are independent, which implies that the likelihood can be decomposed into a product of dimension-wise probabilities:

$$P(\mathbf{x}|c_i) = P(x_1, x_2, \ldots, x_d|c_i) = \prod_{j=1}^{d} P(x_j|c_i)$$

The likelihood for class c_i, for dimension X_j, is given as

$$P(x_j|c_i) \propto f(x_j|\hat{\mu}_{ij}, \hat{\sigma}_{ij}^2) = \frac{1}{\sqrt{2\pi}\hat{\sigma}_{ij}} \exp\left\{-\frac{(x_j - \hat{\mu}_{ij})^2}{2\hat{\sigma}_{ij}^2}\right\}$$

where $\hat{\mu}_{ij}$ and $\hat{\sigma}_{ij}^2$ denote the estimated mean and variance for attribute X_j, for class c_i.

Naive Bayes Classifier: Numeric Attributes

The naive assumption corresponds to setting all the covariances to zero in $\widehat{\boldsymbol{\Sigma}}_i$, that is,

$$\boldsymbol{\Sigma}_i = \begin{pmatrix} \sigma_{i1}^2 & 0 & \ldots & 0 \\ 0 & \sigma_{i2}^2 & \ldots & 0 \\ \vdots & \vdots & \ddots & \\ 0 & 0 & \ldots & \sigma_{id}^2 \end{pmatrix}$$

The naive Bayes classifier thus uses the sample mean $\hat{\boldsymbol{\mu}}_i = (\hat{\mu}_{i1}, \ldots, \hat{\mu}_{id})^T$ and a *diagonal* sample covariance matrix $\widehat{\boldsymbol{\Sigma}}_i = diag(\sigma_{i1}^2, \ldots, \sigma_{id}^2)$ for each class c_i. In total $2d$ parameters have to be estimated, corresponding to the sample mean and sample variance for each dimension X_j.

Naive Bayes Algorithm

NAIVEBAYES ($\mathbf{D} = \{(\mathbf{x}_j, y_j)\}_{j=1}^{n}$):

for $i = 1, \ldots, k$ **do**
  $\mathbf{D}_i \leftarrow \{\mathbf{x}_j \mid y_j = c_i, j = 1, \ldots, n\}$ // class-specific subsets
  $n_i \leftarrow |\mathbf{D}_i|$ // cardinality
  $\hat{P}(c_i) \leftarrow n_i/n$ // prior probability
  $\hat{\boldsymbol{\mu}}_i \leftarrow \frac{1}{n_i} \sum_{\mathbf{x}_j \in \mathbf{D}_i} \mathbf{x}_j$ // mean
  $\mathbf{Z}_i = \mathbf{D}_i - \mathbf{1} \cdot \hat{\boldsymbol{\mu}}_i^T$ // centered data for class c_i
  for $j = 1, .., d$ **do** // class-specific variance for X_j
    $\hat{\sigma}_{ij}^2 \leftarrow \frac{1}{n_i} Z_{ij}^T Z_{ij}$ // variance
  $\hat{\boldsymbol{\sigma}}_i = \left(\hat{\sigma}_{i1}^2, \ldots, \hat{\sigma}_{id}^2\right)^T$ // class-specific attribute variances
return $\hat{P}(c_i), \hat{\boldsymbol{\mu}}_i, \hat{\boldsymbol{\sigma}}_i$ for all $i = 1, \ldots, k$

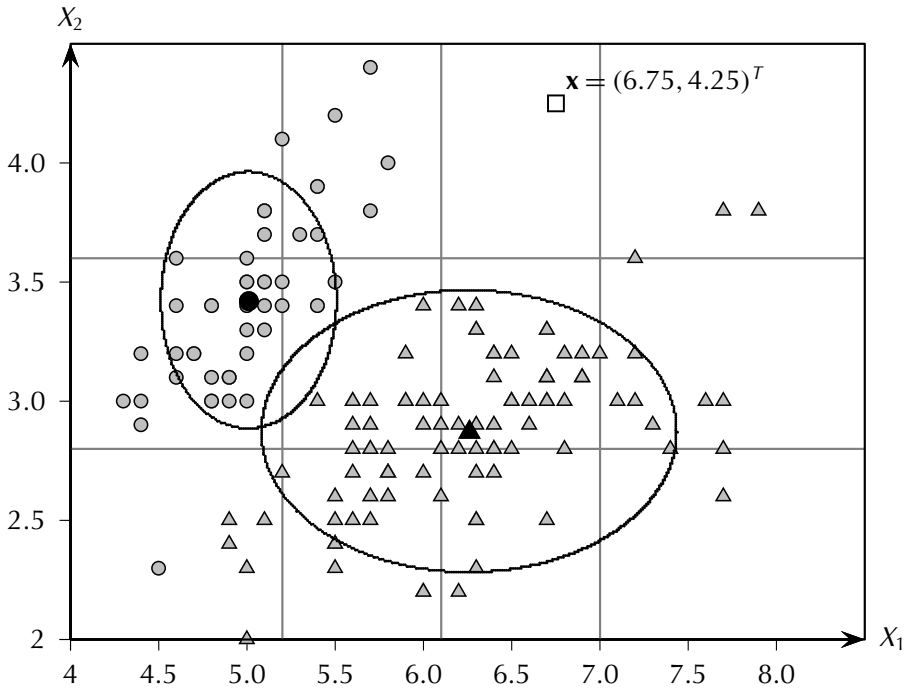
TESTING (**x** and $\hat{P}(c_i)$, $\hat{\boldsymbol{\mu}}_i$, $\hat{\boldsymbol{\sigma}}_i$, **for all** $i \in [1, k]$):

$$\hat{y} \leftarrow \arg\max_{c_i} \left\{ \hat{P}(c_i) \prod_{j=1}^{d} f(x_j | \hat{\mu}_{ij}, \hat{\sigma}_{ij}^2) \right\}$$

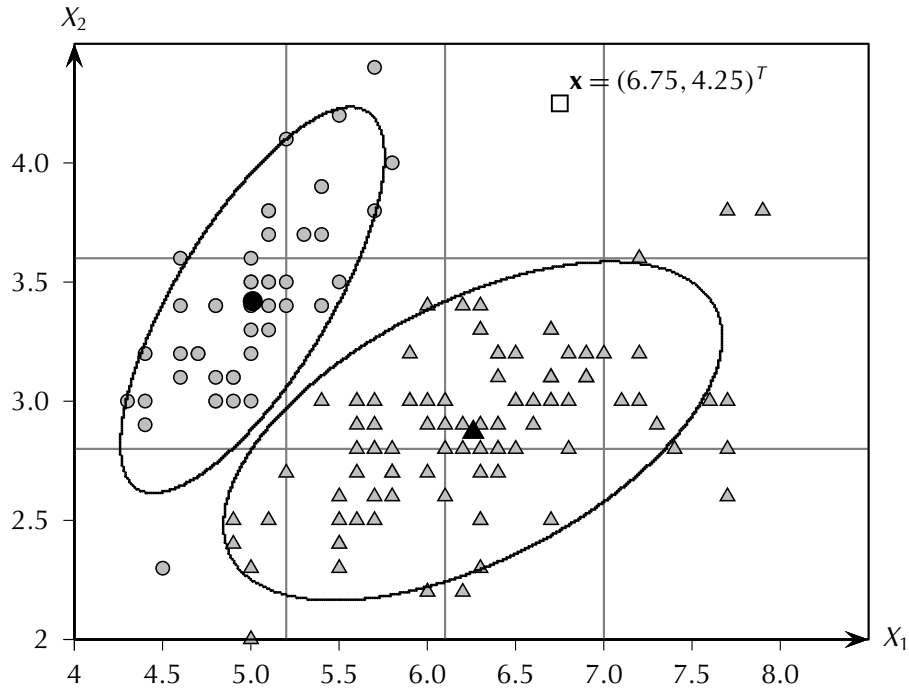
return $\hat{y}$

Naive Bayes versus Full Bayes Classifier: Iris 2D Data

X_1:sepal length versus X_2:sepal width



(a) Naive Bayes



(b) Full Bayes

Naive Bayes: Categorical Attributes

The independence assumption leads to a simplification of the joint probability mass function

$$P(\mathbf{x}|c_i) = \prod_{j=1}^{d} P(x_j|c_i) = \prod_{j=1}^{d} f(\mathbf{X}_j = \mathbf{e}_{jr_j}|\ c_i)$$

where $f(\mathbf{X}_j = \mathbf{e}_{jr_j}|c_i)$ is the probability mass function for $\mathbf{X}_j$, which can be estimated from $\mathbf{D}_i$ as follows:

$$\hat{f}(\mathbf{v}_j|c_i) = \frac{n_i(\mathbf{v}_j)}{n_i}$$

where $n_i(\mathbf{v}_j)$ is the observed frequency of the value $\mathbf{v}_j = \mathbf{e}_j r_j$ corresponding to the r_jth categorical value a_{jr_j} for the attribute X_j for class c_i.

If the count is zero, we can use the pseudo-count method to obtain a prior probability. The adjusted estimates with pseudo-counts are given as

$$\hat{f}(\mathbf{v}_j|c_i) = \frac{n_i(\mathbf{v}_j) + 1}{n_i + m_j}$$

where $m_j = |dom(X_j)|$.

Data Mining

Paradigms

- Combinatorial
- Probabilistic
- **Algebraic**
- Graph-based

Domain

Problem is modeled using linear algebra, enabling several existing algebraic models and algorithms.

Task

Determine the best models and their parameters, according to an optimization metric.

Strategies

- Direct
- Iterative

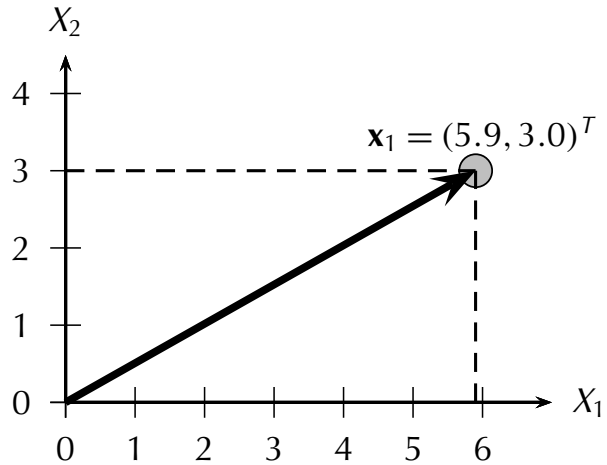
Data: Algebraic and Geometric View

For numeric data matrix **D**, each row or point is a d-dimensional column vector:

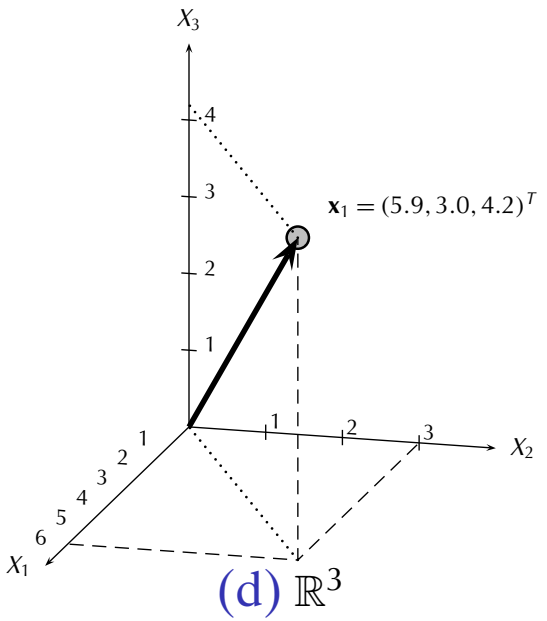
$$\mathbf{x}_i = \begin{pmatrix} x_{i1} \\ x_{i2} \\ \vdots \\ x_{id} \end{pmatrix} = \begin{pmatrix} x_{i1} & x_{i2} & \cdots & x_{id} \end{pmatrix}^T \in \mathbb{R}^d$$

whereas each column or attribute is a n-dimensional column vector:

$$X_j = \begin{pmatrix} x_{1j} & x_{2j} & \cdots & x_{nj} \end{pmatrix}^T \in \mathbb{R}^n$$



(c) $\mathbb{R}^2$

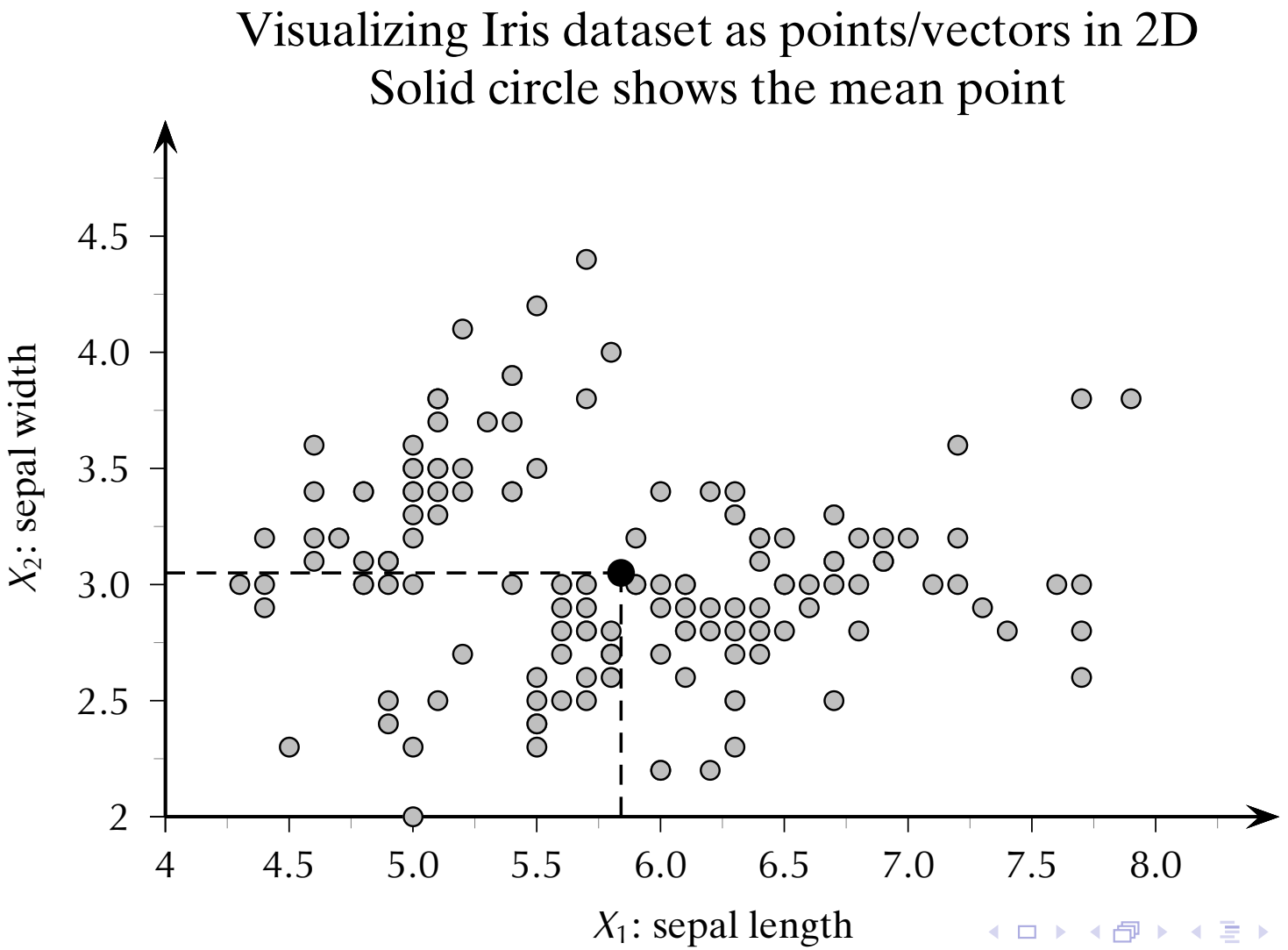


(d) $\mathbb{R}^3$

Figure: Projections of $\mathbf{x}_1 = (5.9, 3.0, 4.2, 1.5)^T$ in 2D and 3D

Scatterplot: 2D Iris Dataset

`sepal length` versus `sepal width`.



Numeric Data Matrix

If all attributes are numeric, then the data matrix $\mathbf{D}$ is an $n \times d$ matrix, or equivalently a set of n row vectors $\mathbf{x}_i^T \in \mathbb{R}^d$ or a set of d column vectors $X_j \in \mathbb{R}^n$

$$\mathbf{D} = \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1d} \\ x_{21} & x_{22} & \cdots & x_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nd} \end{pmatrix} = \begin{pmatrix} — \mathbf{x}_1^T — \\ — \mathbf{x}_2^T — \\ \vdots \\ — \mathbf{x}_n^T — \end{pmatrix} = \begin{pmatrix} | & | & & | \\ X_1 & X_2 & \cdots & X_d \\ | & | & & | \end{pmatrix}$$

The *mean* of the data matrix $\mathbf{D}$ is the average of all the points: $mean(\mathbf{D}) = \boldsymbol{\mu} = \frac{1}{n}\sum_{i=1}^{n} \mathbf{x}_i$

The *centered data matrix* is obtained by subtracting the mean from all the points:

$$\mathbf{Z} = \mathbf{D} - \mathbf{1} \cdot \boldsymbol{\mu}^T = \begin{pmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \vdots \\ \mathbf{x}_n^T \end{pmatrix} - \begin{pmatrix} \boldsymbol{\mu}^T \\ \boldsymbol{\mu}^T \\ \vdots \\ \boldsymbol{\mu}^T \end{pmatrix} = \begin{pmatrix} \mathbf{x}_1^T - \boldsymbol{\mu}^T \\ \mathbf{x}_2^T - \boldsymbol{\mu}^T \\ \vdots \\ \mathbf{x}_n^T - \boldsymbol{\mu}^T \end{pmatrix} = \begin{pmatrix} \mathbf{z}_1^T \\ \mathbf{z}_2^T \\ \vdots \\ \mathbf{z}_n^T \end{pmatrix} \tag{2}$$

where $\mathbf{z}_i = \mathbf{x}_i - \boldsymbol{\mu}$ is a centered point, and $\mathbf{1} \in \mathbb{R}^n$ is the vector of ones.

Norm, Distance and Angle

Given two points $\mathbf{a}, \mathbf{b} \in \mathbb{R}^m$, their *dot product* is defined as the scalar

$$\mathbf{a}^T\mathbf{b} = a_1 b_1 + a_2 b_2 + \cdots + a_m b_m = \sum_{i=1}^{m} a_i b_i$$

The *Euclidean norm* or *length* of a vector $\mathbf{a}$ is defined as

$$\|\mathbf{a}\| = \sqrt{\mathbf{a}^T\mathbf{a}} = \sqrt{\sum_{i=1}^{m} a_i^2}$$

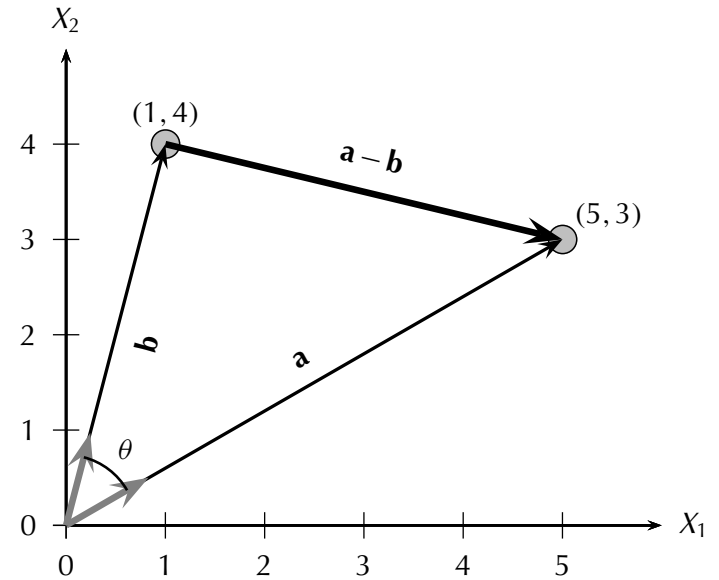
The *unit vector* in the direction of $\mathbf{a}$ is $\mathbf{u} = \frac{\mathbf{a}}{\|\mathbf{a}\|}$ with $\|\mathbf{a}\| = 1$.

Distance between $\mathbf{a}$ and $\mathbf{b}$ is given as

$$\|\mathbf{a} - \mathbf{b}\| = \sqrt{\sum_{i=1}^{m} (a_i - b_i)^2}$$

Angle between $\mathbf{a}$ and $\mathbf{b}$ is given as

$$\cos\theta = \frac{\mathbf{a}^T\mathbf{b}}{\|\mathbf{a}\|\,\|\mathbf{b}\|} = \left(\frac{\mathbf{a}}{\|\mathbf{a}\|}\right)^T \left(\frac{\mathbf{b}}{\|\mathbf{b}\|}\right)$$



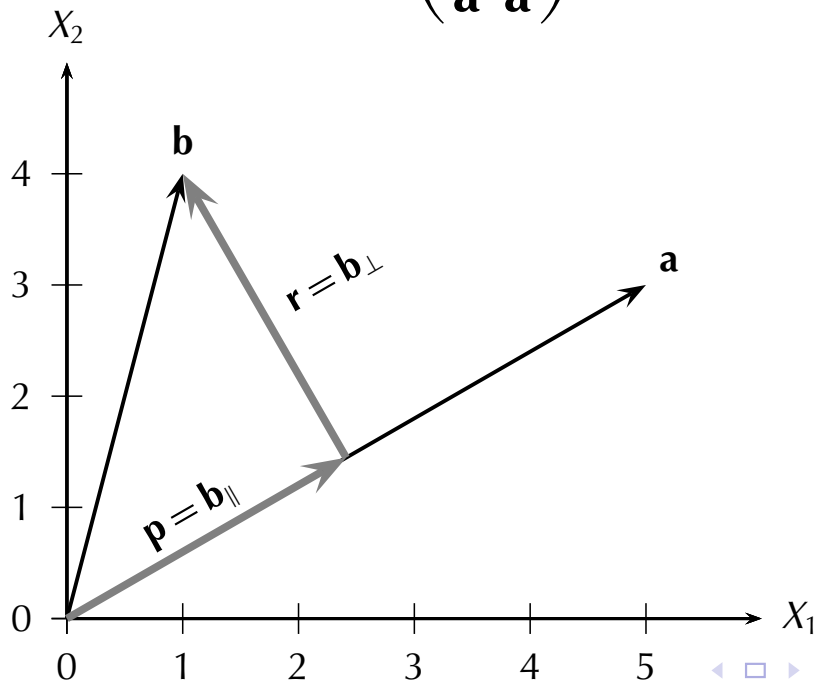
Orthogonal Projection

Two vectors $\mathbf{a}$ and $\mathbf{b}$ are *orthogonal* iff $\mathbf{a}^T\mathbf{b} = 0$, i.e., the angle between them is 90°. Orthogonal projection of $\mathbf{b}$ on $\mathbf{a}$ comprises the vector $\mathbf{p} = \mathbf{b}_\parallel$ parallel to $\mathbf{a}$, and $\mathbf{r} = \mathbf{b}_\perp$ perpendicular or orthogonal to $\mathbf{a}$, given as

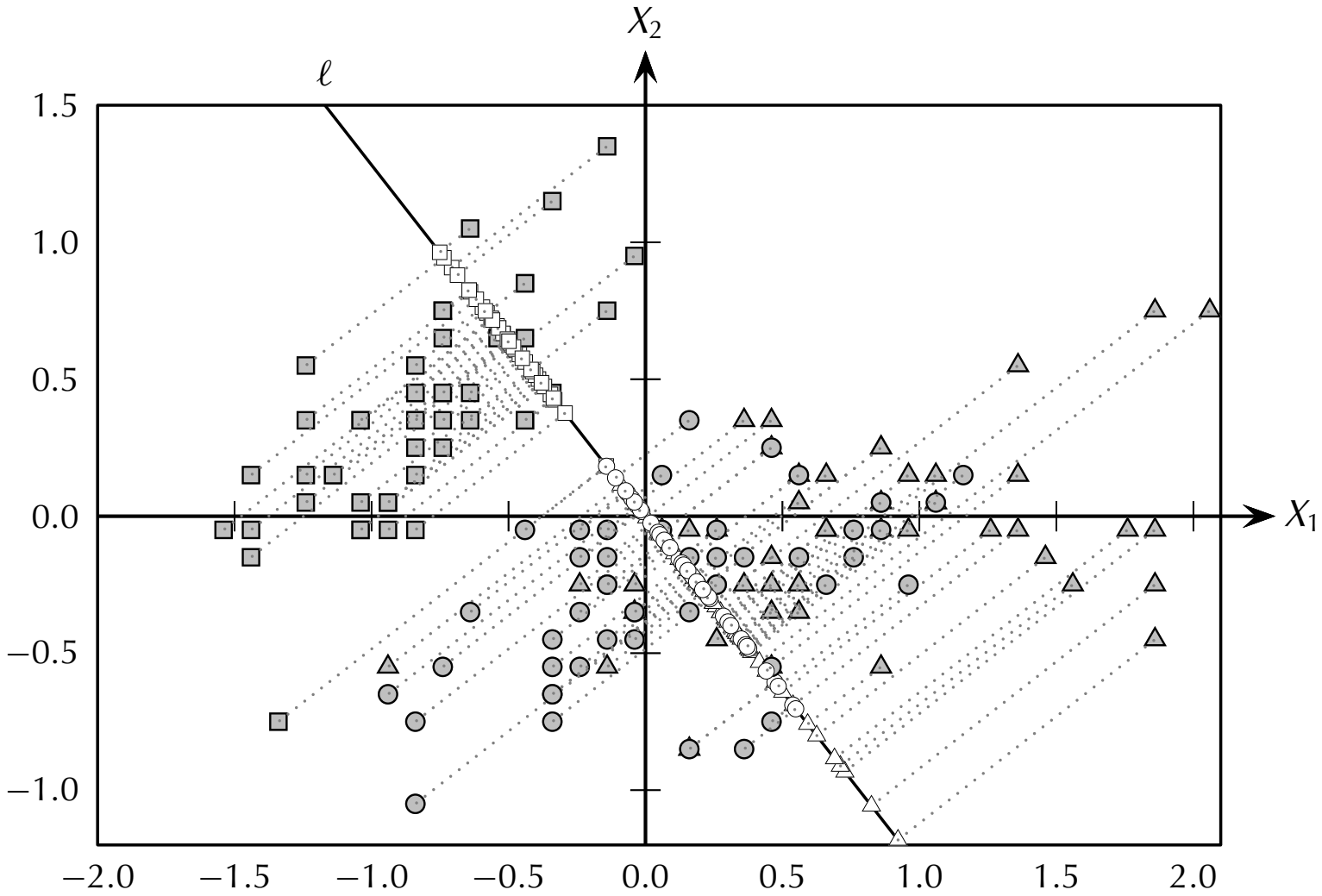
$$\mathbf{b} = \mathbf{b}_\parallel + \mathbf{b}_\perp = \mathbf{p} + \mathbf{r}$$

where

$$\mathbf{p} = \mathbf{b}_\parallel = \left(\frac{\mathbf{a}^T\mathbf{b}}{\mathbf{a}^T\mathbf{a}}\right)\mathbf{a}$$



Projection of Centered Iris Data Onto a Line ℓ.



- **Principal Component Analysis**
- Support Vector Machines

Dimensionality Reduction

The goal of dimensionality reduction is to find a lower dimensional representation of the data matrix $\mathbf{D}$ to avoid the curse of dimensionality.

Given $n \times d$ data matrix, each point $\mathbf{x}_i = (x_{i1}, x_{i2}, \ldots, x_{id})^T$ is a vector in the ambient d-dimensional vector space spanned by the d standard basis vectors $\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_d$.

Given any other set of d orthonormal vectors $\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_d$ we can re-express each point $\mathbf{x}$ as

$$\mathbf{x} = a_1\mathbf{u}_1 + a_2\mathbf{u}_2 + \cdots + a_d\mathbf{u}_d$$

where $\mathbf{a} = (a_1, a_2, \ldots, a_d)^T$ represents the coordinates of $\mathbf{x}$ in the new basis. More compactly:

$$\mathbf{x} = \mathbf{U}\mathbf{a}$$

where $\mathbf{U}$ is the $d \times d$ orthogonal matrix, whose ith column comprises the ith basis vector $\mathbf{u}_i$. Thus $\mathbf{U}^{-1} = \mathbf{U}^T$, and we have

$$\mathbf{a} = \mathbf{U}^T\mathbf{x}$$

Optimal Basis: Projection in Lower Dimensional Space

There are potentially infinite choices for the orthonormal basis vectors. Our goal is to choose an *optimal* basis that preserves essential information about $\mathbf{D}$.

We are interested in finding the optimal r-dimensional representation of $\mathbf{D}$, with $r \ll d$. Projection of $\mathbf{x}$ onto the first r basis vectors is given as

$$\mathbf{x}' = a_1\mathbf{u}_1 + a_2\mathbf{u}_2 + \cdots + a_r\mathbf{u}_r = \sum_{i=1}^{r} a_i\mathbf{u}_i = \mathbf{U}_r\mathbf{a}_r$$

where $\mathbf{U}_r$ and $\mathbf{a}_r$ comprises the r basis vectors and coordinates, respv. Also, restricting $\mathbf{a} = \mathbf{U}^T\mathbf{x}$ to r terms, we have

$$\mathbf{a}_r = \mathbf{U}_r^T\mathbf{x}$$

The r-dimensional projection of $\mathbf{x}$ is thus given as:

$$\mathbf{x}' = \mathbf{U}_r\mathbf{U}_r^T\mathbf{x} = \mathbf{P}_r\mathbf{x}$$

where $\mathbf{P}_r = \mathbf{U}_r\mathbf{U}_r^T = \sum_{i=1}^{r} \mathbf{u}_i\mathbf{u}_i^T$ is the *orthogonal projection matrix* for the subspace spanned by the first r basis vectors.

Optimal Basis: Error Vector

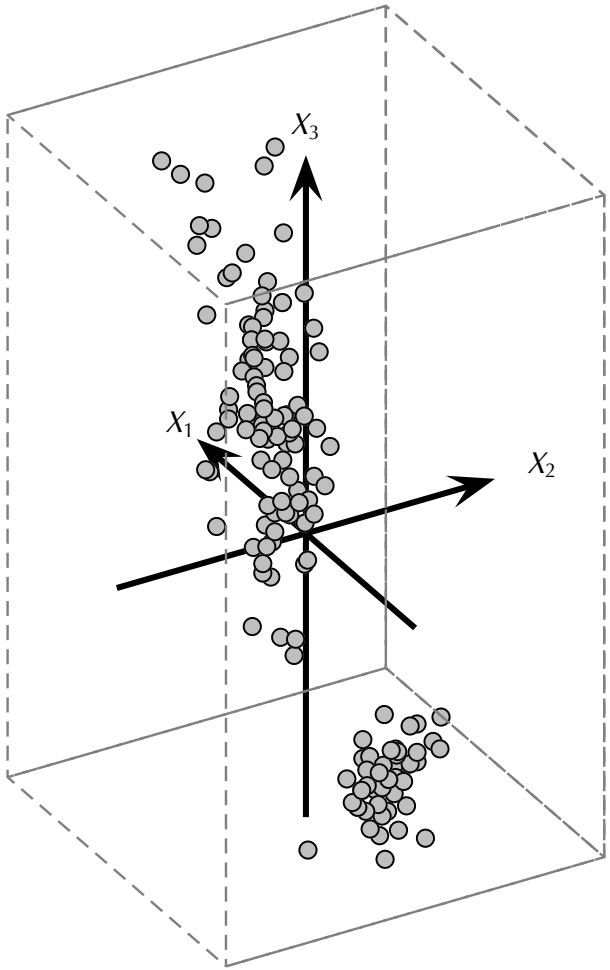
Given the projected vector $\mathbf{x}' = \mathbf{P}_r\mathbf{x}$, the corresponding *error vector*, is the projection onto the remaining $d - r$ basis vectors

$$\boldsymbol{\epsilon} = \sum_{i=r+1}^{d} a_i \mathbf{u}_i = \mathbf{x} - \mathbf{x}'$$

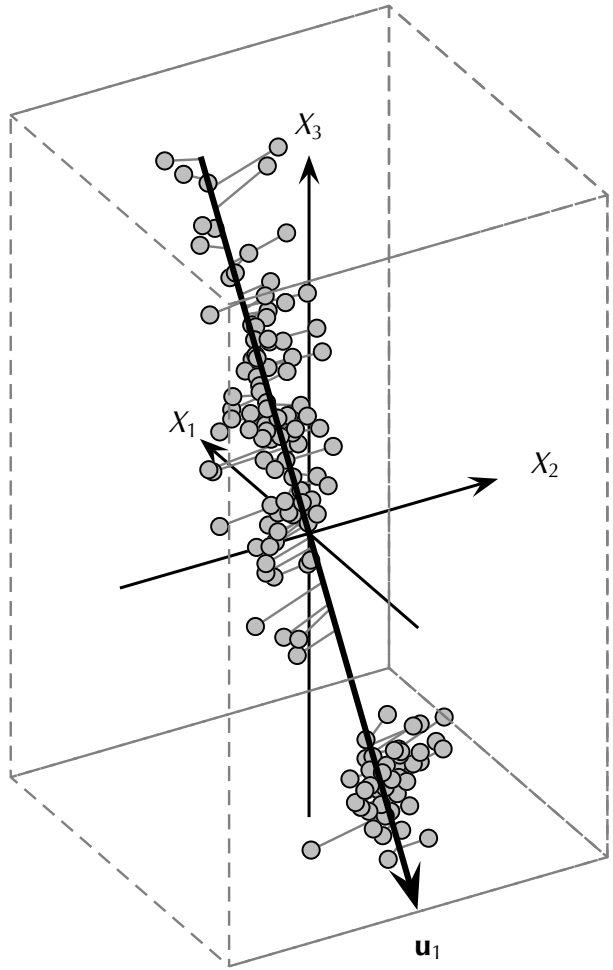
The error vector $\boldsymbol{\epsilon}$ is orthogonal to $\mathbf{x}'$.

The goal of dimensionality reduction is to seek an r-dimensional basis that gives the best possible approximation $\mathbf{x}'_i$ over all the points $\mathbf{x}_i \in \mathbf{D}$. Alternatively, we seek to minimize the error $\boldsymbol{\epsilon}_i = \mathbf{x}_i - \mathbf{x}'_i$ over all the points.

Iris Data: Optimal One-dimensional Basis

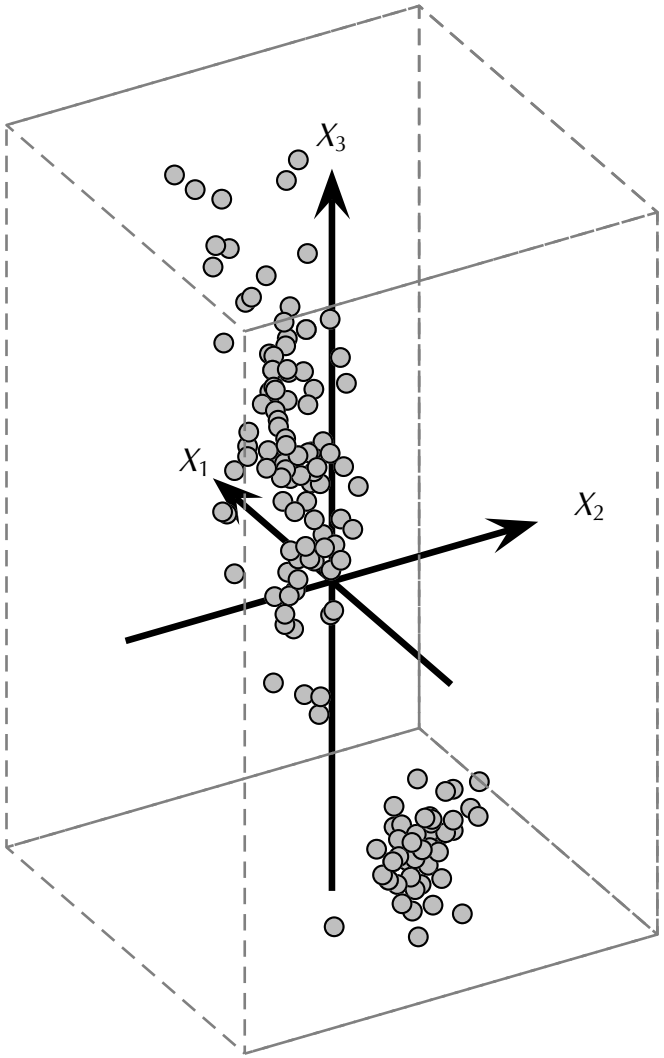


Iris Data: 3D

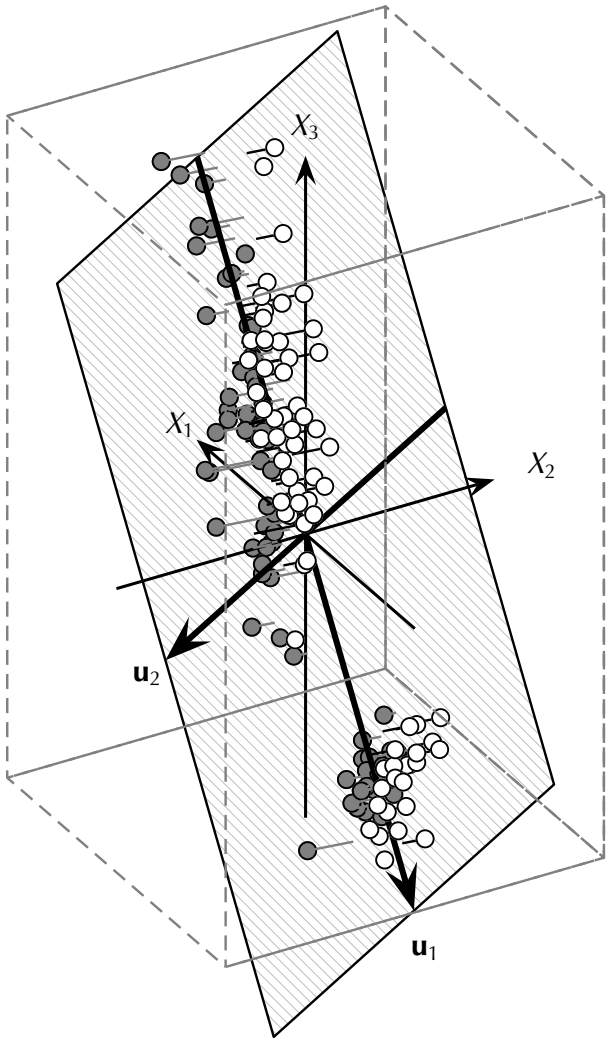


Optimal 1D Basis

Iris Data: Optimal 2D Basis



Iris Data (3D)



Optimal 2D Basis

Principal Component Analysis

Principal Component Analysis (PCA) is a technique that seeks a r-dimensional basis that best captures the variance in the data.

The direction with the largest projected variance is called the first principal component.

The orthogonal direction that captures the second largest projected variance is called the second principal component, and so on.

The direction that maximizes the variance is also the one that minimizes the mean squared error.

Principal Component: Direction of Most Variance

We seek to find the unit vector $\mathbf{u}$ that maximizes the projected variance of the points. Let $\mathbf{D}$ be centered, and let $\mathbf{\Sigma}$ be its covariance matrix.

The projection of $\mathbf{x}_i$ on $\mathbf{u}$ is given as

$$\mathbf{x}_i' = \left(\frac{\mathbf{u}^T\mathbf{x}_i}{\mathbf{u}^T\mathbf{u}}\right)\mathbf{u} = (\mathbf{u}^T\mathbf{x}_i)\mathbf{u} = a_i\mathbf{u}$$

Across all the points, the projected variance along $\mathbf{u}$ is

$$\sigma_{\mathbf{u}}^2 = \frac{1}{n}\sum_{i=1}^{n}(a_i - \mu_{\mathbf{u}})^2 = \frac{1}{n}\sum_{i=1}^{n}\mathbf{u}^T\left(\mathbf{x}_i\mathbf{x}_i^T\right)\mathbf{u} = \mathbf{u}^T\left(\frac{1}{n}\sum_{i=1}^{n}\mathbf{x}_i\mathbf{x}_i^T\right)\mathbf{u} = \mathbf{u}^T\mathbf{\Sigma}\mathbf{u}$$

We have to find the optimal basis vector $\mathbf{u}$ that maximizes the projected variance $\sigma_{\mathbf{u}}^2 = \mathbf{u}^T\mathbf{\Sigma}\mathbf{u}$, subject to the constraint that $\mathbf{u}^T\mathbf{u} = 1$. The maximization objective is given as

$$\max_{\mathbf{u}} J(\mathbf{u}) = \mathbf{u}^T\mathbf{\Sigma}\mathbf{u} - \alpha(\mathbf{u}^T\mathbf{u} - 1)$$

Principal Component: Direction of Most Variance

Given the objective $\max_{\mathbf{u}} J(\mathbf{u}) = \mathbf{u}^T\mathbf{\Sigma}\mathbf{u} - \alpha(\mathbf{u}^T\mathbf{u} - 1)$, we solve it by setting the derivative of $J(\mathbf{u})$ with respect to $\mathbf{u}$ to the zero vector, to obtain

$$\frac{\partial}{\partial \mathbf{u}}\left(\mathbf{u}^T\mathbf{\Sigma}\mathbf{u} - \alpha(\mathbf{u}^T\mathbf{u} - 1)\right) = \mathbf{0}$$

$$\text{that is, } 2\mathbf{\Sigma}\mathbf{u} - 2\alpha\mathbf{u} = \mathbf{0} \text{which implies } \mathbf{\Sigma}\mathbf{u} = \alpha\mathbf{u}$$

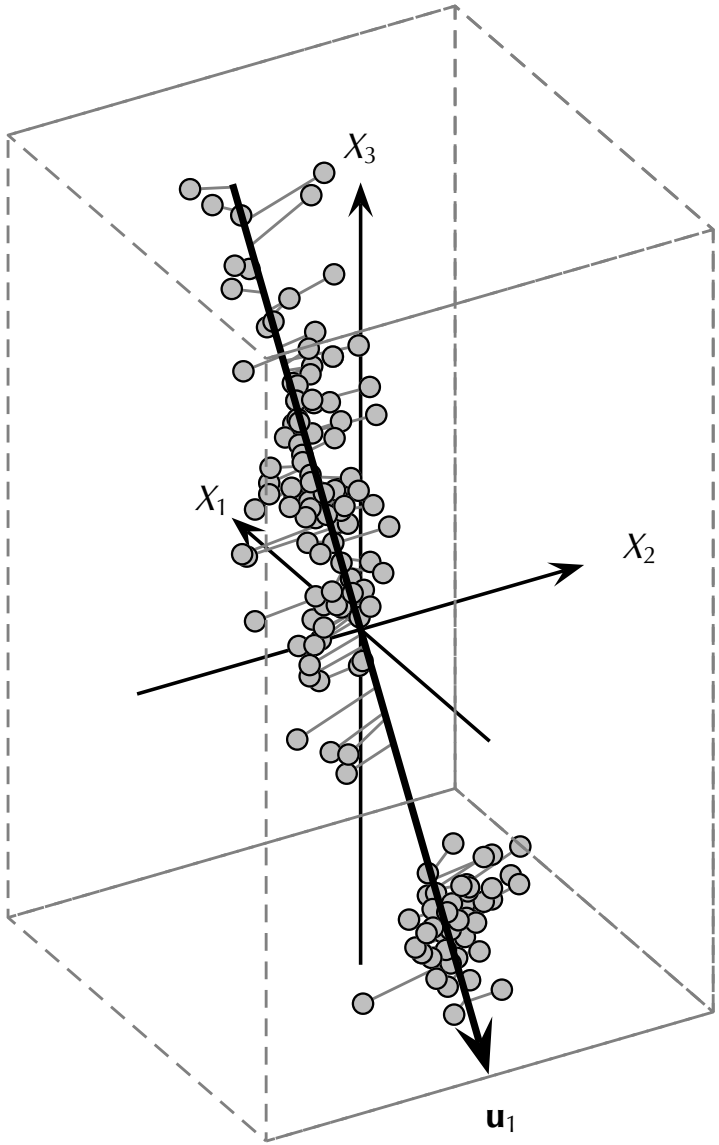
Thus α is an eigenvalue of the covariance matrix $\mathbf{\Sigma}$, with the associated eigenvector $\mathbf{u}$.

Taking the dot product with $\mathbf{u}$ on both sides, we have

$$\sigma_{\mathbf{u}}^2 = \mathbf{u}^T\mathbf{\Sigma}\mathbf{u}\mathbf{u}^T\alpha\mathbf{u} = \alpha\mathbf{u}^T\mathbf{u} = \alpha$$

To maximize the projected variance $\sigma_{\mathbf{u}}^2$, we thus choose the largest eigenvalue λ_1 of $\mathbf{\Sigma}$, and the dominant eigenvector $\mathbf{u}_1$ specifies the direction of most variance, also called the *first principal component*.

Iris Data: First Principal Component



Minimum Squared Error Approach

The direction that maximizes the projected variance is also the one that minimizes the average squared error.

The mean squared error (MSE) optimization condition is

$$MSE(\mathbf{u}) = \frac{1}{n}\sum_{i=1}^{n}\|\boldsymbol{\epsilon}_i\|^2 = \frac{1}{n}\sum_{i=1}^{n}\|\mathbf{x}_i - \mathbf{x}_i'\|^2 = \sum_{i=1}^{n}\frac{\|\mathbf{x}_i\|^2}{n} - \mathbf{u}^T\boldsymbol{\Sigma}\mathbf{u}$$

Since the first term is fixed for a dataset $\mathbf{D}$, we see that the direction $\mathbf{u}_1$ that maximizes the variance is also the one that minimizes the MSE.

Further, we have

$$\sum_{i=1}^{n}\frac{\|\mathbf{x}_i\|^2}{n} - \mathbf{u}^T\boldsymbol{\Sigma}\mathbf{u} = var(\mathbf{D}) = tr(\boldsymbol{\Sigma}) = \sum_{i=1}^{d}\sigma_i^2$$

Thus, for the direction $\mathbf{u}_1$ that minimizes MSE, we have

$$MSE(\mathbf{u}_1) = var(\mathbf{D}) - \mathbf{u}_1^T\boldsymbol{\Sigma}\mathbf{u}_1 = var(\mathbf{D}) - \lambda_1$$

Best 2-dimensional Approximation

The best 2D subspace that captures the most variance in $\mathbf{D}$ comprises the eigenvectors $\mathbf{u}_1$ and $\mathbf{u}_2$ corresponding to the largest and second largest eigenvalues λ_1 and λ_2, respv.

Let $\mathbf{U}_2 = \begin{pmatrix} \mathbf{u}_1 & \mathbf{u}_2 \end{pmatrix}$ be the matrix whose columns correspond to the two principal components. Given the point $\mathbf{x}_i \in \mathbf{D}$ its projected coordinates are computed as follows:

$$\mathbf{a}_i = \mathbf{U}_2^T \mathbf{x}_i$$

Let $\mathbf{A}$ denote the projected 2D dataset. The total projected variance for $\mathbf{A}$ is given as

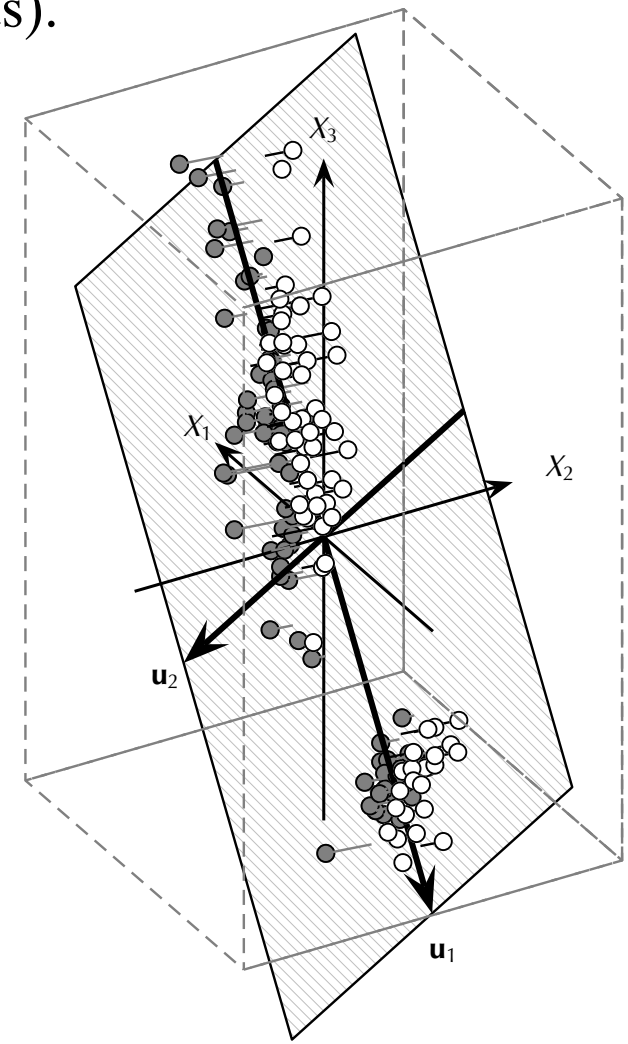
$$var(\mathbf{A}) = \mathbf{u}_1^T \mathbf{\Sigma} \mathbf{u}_1 + \mathbf{u}_2^T \mathbf{\Sigma} \mathbf{u}_2 = \mathbf{u}_1^T \lambda_1 \mathbf{u}_1 + \mathbf{u}_2^T \lambda_2 \mathbf{u}_2 = \lambda_1 + \lambda_2$$

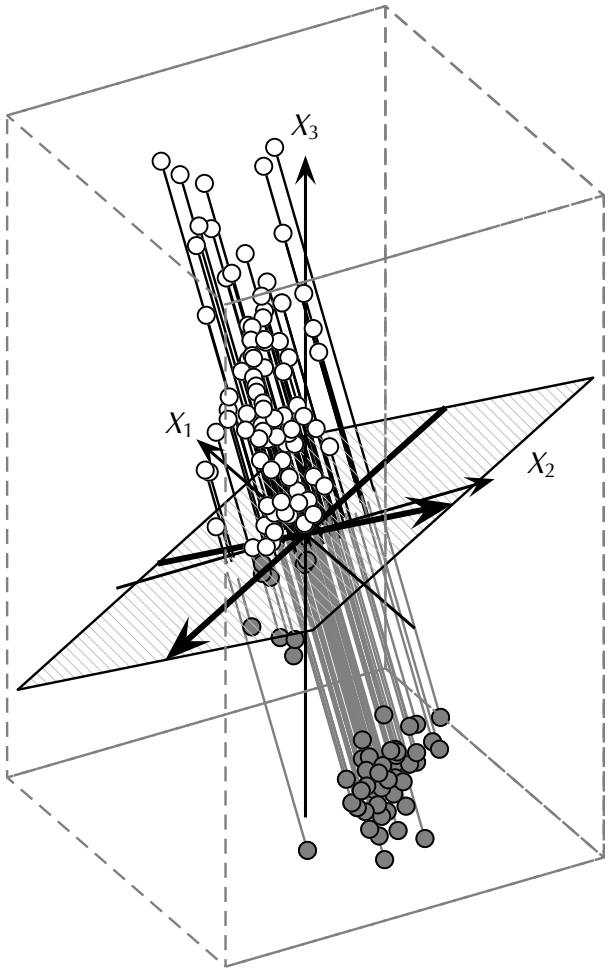
The first two principal components also minimize the mean square error objective, since

$$MSE = \frac{1}{n} \sum_{i=1}^{n} \left\| \mathbf{x}_i - \mathbf{x}_i' \right\|^2 = var(\mathbf{D}) - \frac{1}{n} \sum_{i=1}^{n} \left(\mathbf{x}_i^T \mathbf{P}_2 \mathbf{x}_i \right) = var(\mathbf{D}) - var(\mathbf{A})$$

Optimal and Non-optimal 2D Approximations

The optimal subspace maximizes the variance, and minimizes the squared error, whereas the nonoptimal subspace captures less variance, and has a high mean squared error value, as seen from the lengths of the error vectors (line segments).





Best *r*-dimensional Approximation

To find the best r-dimensional approximation to $\mathbf{D}$, we compute the eigenvalues of $\mathbf{\Sigma}$. Because $\mathbf{\Sigma}$ is positive semidefinite, its eigenvalues are non-negative

$$\lambda_1 \geq \lambda_2 \geq \cdots \lambda_r \geq \lambda_{r+1} \cdots \geq \lambda_d \geq 0$$

We select the r largest eigenvalues, and their corresponding eigenvectors to form the best r-dimensional approximation.

Total Projected Variance: Let $\mathbf{U}_r = \begin{pmatrix} \mathbf{u}_1 & \cdots & \mathbf{u}_r \end{pmatrix}$ be the r-dimensional basis vector matrix, withe the projection matrix given as $\mathbf{P}_r = \mathbf{U}_r\mathbf{U}_r^T = \sum_{i=1}^{r} \mathbf{u}_i\mathbf{u}_i^T$.

Let $\mathbf{A}$ denote the dataset formed by the coordinates of the projected points in the r-dimensional subspace. The projected variance is given as

$$var(\mathbf{A}) = \frac{1}{n}\sum_{i=1}^{n} \mathbf{x}_i^T\mathbf{P}_r\mathbf{x}_i = \sum_{i=1}^{r} \mathbf{u}_i^T\mathbf{\Sigma}\mathbf{u}_i = \sum_{i=1}^{r} \lambda_i$$

Mean Squared Error: The mean squared error in r dimensions is

$$MSE = \frac{1}{n}\sum_{i=1}^{n} \left\|\mathbf{x}_i - \mathbf{x}_i'\right\|^2 = var(\mathbf{D}) - \sum_{i=1}^{r} \lambda_i = \sum_{i=1}^{d} \lambda_i - \sum_{i=1}^{r} \lambda_i$$

Choosing the Dimensionality

One criteria for choosing r is to compute the fraction of the total variance captured by the first r principal components, computed as

$$f(r) = \frac{\lambda_1 + \lambda_2 + \cdots + \lambda_r}{\lambda_1 + \lambda_2 + \cdots + \lambda_d} = \frac{\sum_{i=1}^{r} \lambda_i}{\sum_{i=1}^{d} \lambda_i} = \frac{\sum_{i=1}^{r} \lambda_i}{var(\mathbf{D})}$$

Given a certain desired variance threshold, say α, starting from the first principal component, we keep on adding additional components, and stop at the smallest value r, for which $f(r) \geq \alpha$. In other words, we select the fewest number of dimensions such that the subspace spanned by those r dimensions captures at least α fraction (say 0.9) of the total variance.

Principal Component Analysis: Algorithm

PCA $(\mathbf{D}, \alpha)$:

1 $\boldsymbol{\mu} = \frac{1}{n}\sum_{i=1}^{n}\mathbf{x}_i$ // compute mean

2 $\mathbf{Z} = \mathbf{D} - \mathbf{1}\cdot\boldsymbol{\mu}^T$ // center the data

3 $\boldsymbol{\Sigma} = \frac{1}{n}\left(\mathbf{Z}^T\mathbf{Z}\right)$ // compute covariance matrix

4 $(\lambda_1, \lambda_2, \ldots, \lambda_d) = \text{eigenvalues}(\boldsymbol{\Sigma})$ // compute eigenvalues

5 $\mathbf{U} = \begin{pmatrix}\mathbf{u}_1 & \mathbf{u}_2 & \cdots & \mathbf{u}_d\end{pmatrix} = \text{eigenvectors}(\boldsymbol{\Sigma})$ // compute eigenvectors

6 $f(r) = \frac{\sum_{i=1}^{r}\lambda_i}{\sum_{i=1}^{d}\lambda_i}$, for all $r = 1, 2, \ldots, d$ // fraction of total variance

7 Choose smallest r so that $f(r) \geq \alpha$ // choose dimensionality

8 $\mathbf{U}_r = \begin{pmatrix}\mathbf{u}_1 & \mathbf{u}_2 & \cdots & \mathbf{u}_r\end{pmatrix}$ // reduced basis

9 $\mathbf{A} = \{\mathbf{a}_i \mid \mathbf{a}_i = \mathbf{U}_r^T\mathbf{x}_i, \text{for } i = 1, \ldots, n\}$ // reduced dimensionality data

Iris Principal Components

Covariance matrix:

$$\mathbf{\Sigma} = \begin{pmatrix} 0.681 & -0.039 & 1.265 \\ -0.039 & 0.187 & -0.320 \\ 1.265 & -0.32 & 3.092 \end{pmatrix}$$

The eigenvalues and eigenvectors of $\mathbf{\Sigma}$

$$\lambda_1 = 3.662 \qquad \lambda_2 = 0.239 \qquad \lambda_3 = 0.059$$

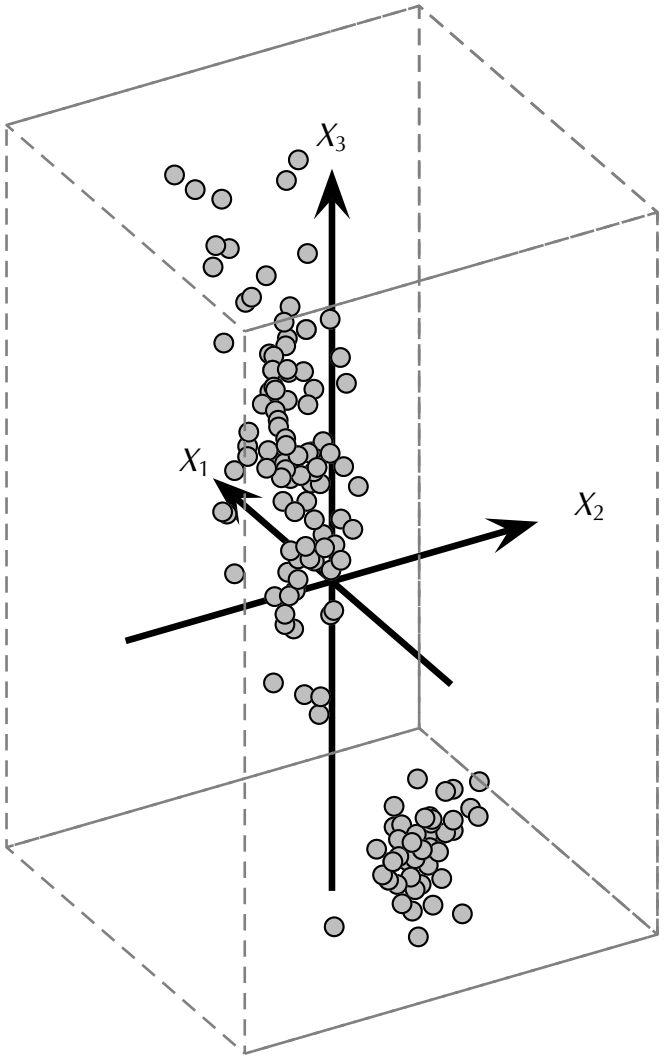
$$\mathbf{u}_1 = \begin{pmatrix} -0.390 \\ 0.089 \\ -0.916 \end{pmatrix} \qquad \mathbf{u}_2 = \begin{pmatrix} -0.639 \\ -0.742 \\ 0.200 \end{pmatrix} \qquad \mathbf{u}_3 = \begin{pmatrix} -0.663 \\ 0.664 \\ 0.346 \end{pmatrix}$$

The total variance is therefore $\lambda_1 + \lambda_2 + \lambda_3 = 3.662 + 0.239 + 0.059 = 3.96$.
The fraction of total variance for different values of r is given as

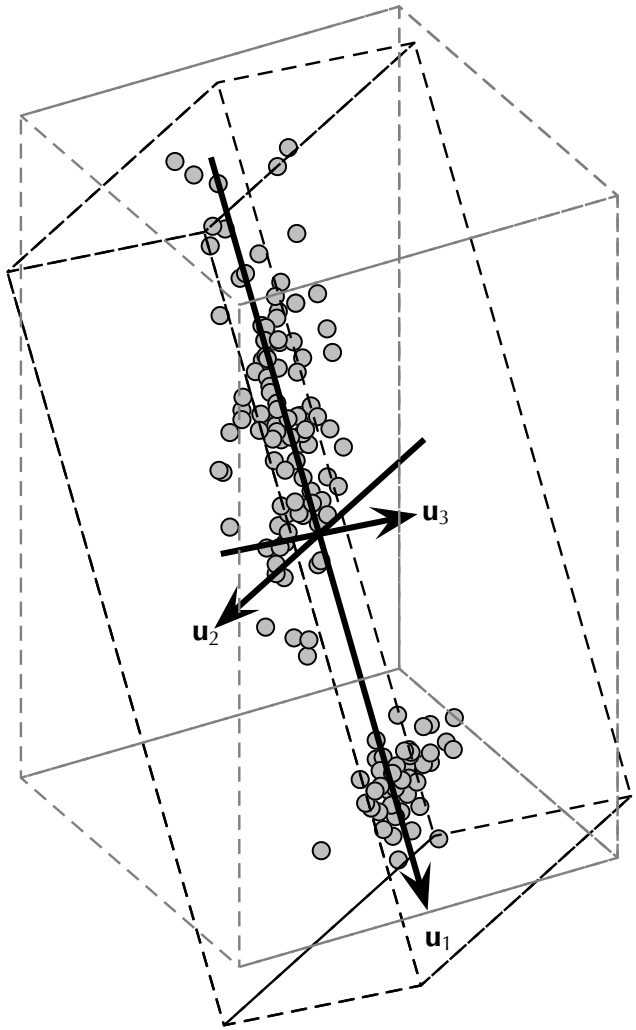
r	1	2	3
$f(r)$	0.925	0.985	1.0

This $r = 2$ PCs are need to capture $\alpha = 0.95$ fraction of variance.

Iris Data: Optimal 3D PC Basis

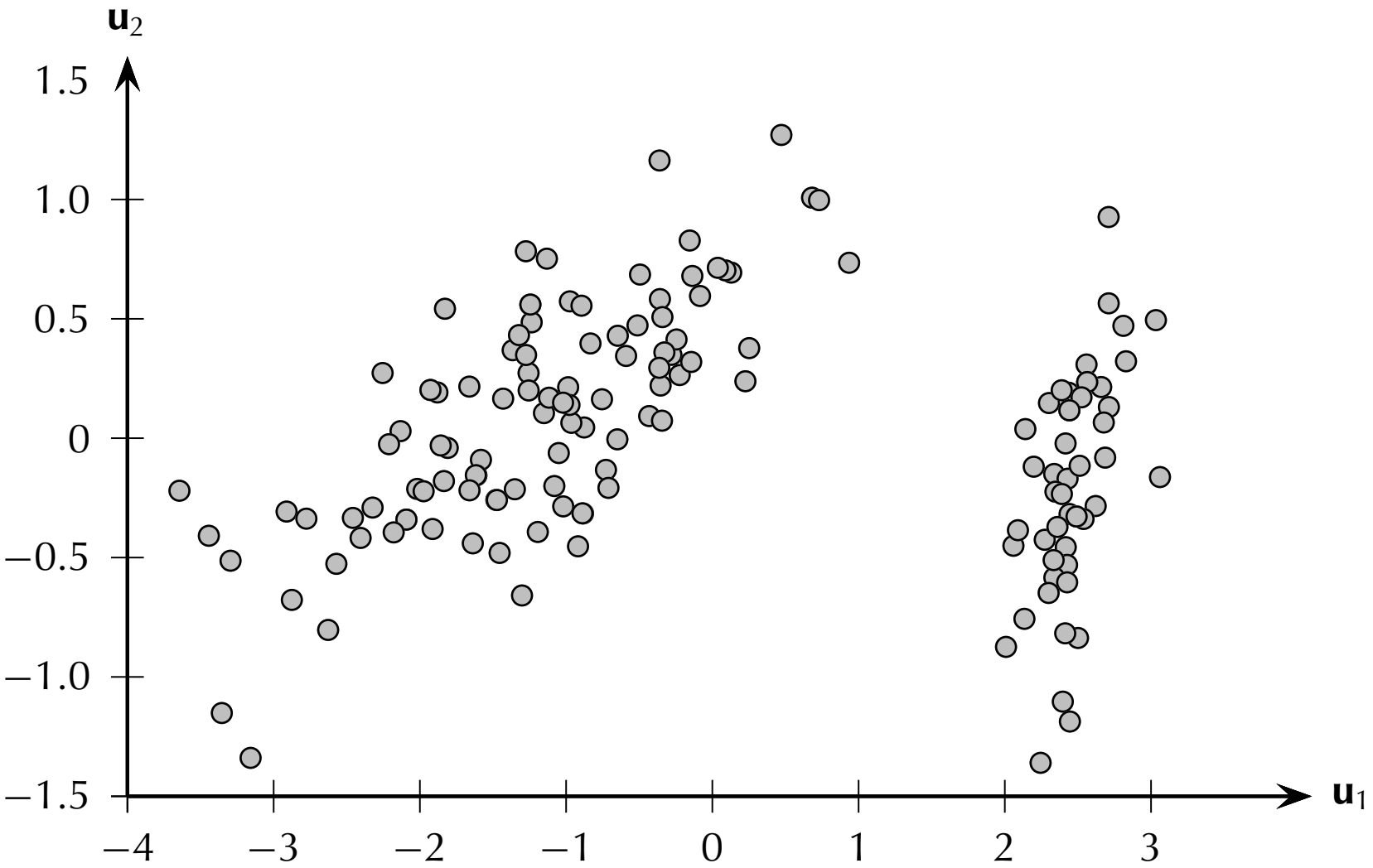


Iris Data (3D)



Optimal 3D Basis

Iris Principal Components: Projected Data (2D)



Geometry of PCA

Geometrically, when $r = d$, PCA corresponds to a orthogonal change of basis, so that the total variance is captured by the sum of the variances along each of the principal directions $\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_d$, and further, all covariances are zero.

Let $\mathbf{U}$ be the $d \times d$ orthogonal matrix $\mathbf{U} = \begin{pmatrix} \mathbf{u}_1 & \mathbf{u}_2 & \cdots & \mathbf{u}_d \end{pmatrix}$, with $\mathbf{U}^{-1} = \mathbf{U}^T$. Let $\mathbf{\Lambda} = diag(\lambda_1, \cdots, \lambda_d)$ be the diagonal matrix of eigenvalues. Each principal component $\mathbf{u}_i$ corresponds to an eigenvector of the covariance matrix $\mathbf{\Sigma}$

$$\mathbf{\Sigma}\mathbf{u}_i = \lambda_i \mathbf{u}_i \text{ for all } 1 \leq i \leq d$$

which can be written compactly in matrix notation:

$$\mathbf{\Sigma}\mathbf{U} = \mathbf{U}\mathbf{\Lambda} \text{ which implies } \mathbf{\Sigma} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^T$$

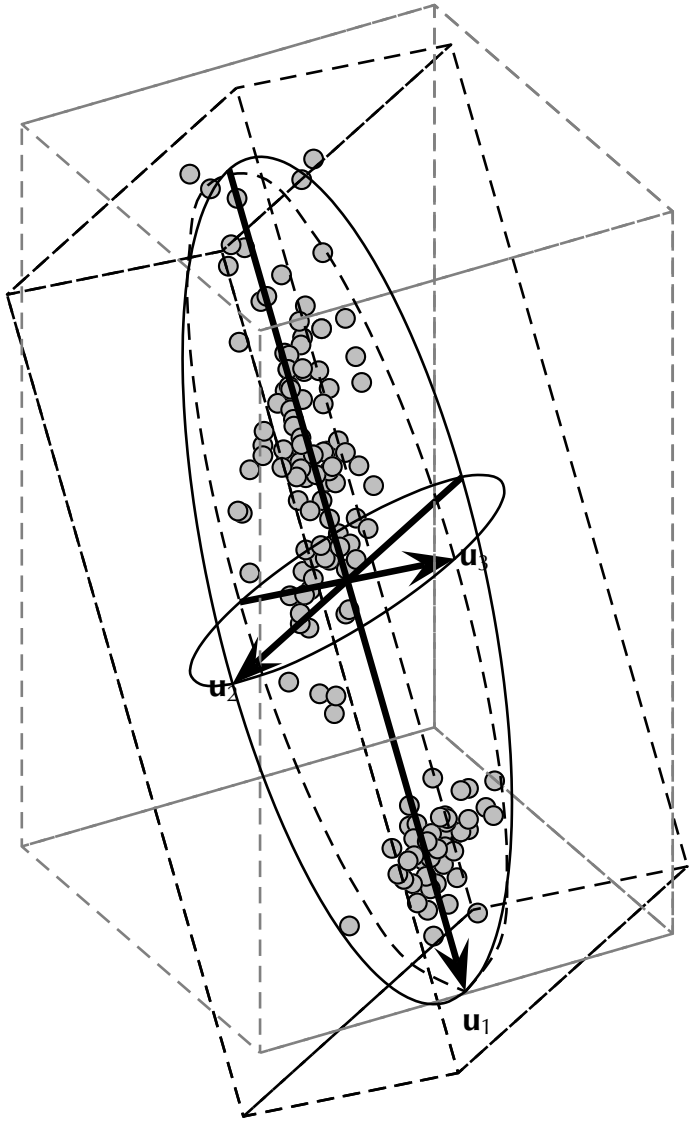
Thus, $\mathbf{\Lambda}$ represents the covariance matrix in the new PC basis.

In the new PC basis, the equation

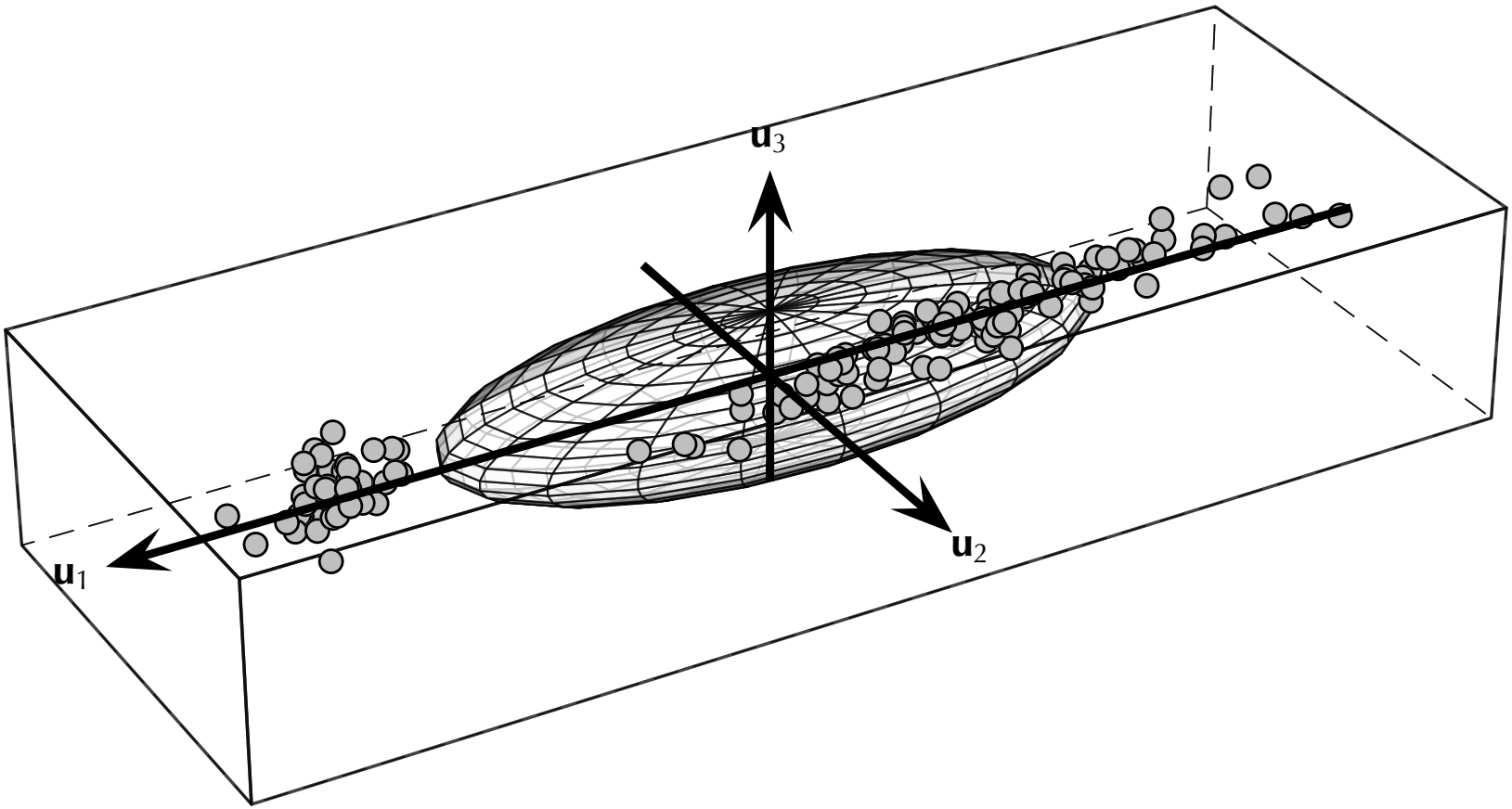
$$\mathbf{x}^T \mathbf{\Sigma}^{-1} \mathbf{x} = 1$$

defines a d-dimensional ellipsoid (or hyper-ellipse). The eigenvectors $\mathbf{u}_i$ of $\mathbf{\Sigma}$, that is, the principal components, are the directions for the principal axes of the ellipsoid. The square roots of the eigenvalues, that is, $\sqrt{\lambda_i}$, give the lengths of the semi-axes.

Iris: Elliptic Contours in Standard Basis



Iris: Axis-Parallel Ellipsoid in PC Basis



- Principal Component Analysis
- **Support Vector Machines**

Hyperplanes

Let $\mathbf{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$ be a classification dataset, with n points in a d-dimensional space. We assume that there are only two class labels, that is, $y_i \in \{+1, -1\}$, denoting the positive and negative classes.

A hyperplane in d dimensions is given as the set of all points $\mathbf{x} \in \mathbb{R}^d$ that satisfy the equation $h(\mathbf{x}) = 0$, where $h(\mathbf{x})$ is the *hyperplane function*:

$$h(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b = w_1 x_1 + w_2 x_2 + \cdots + w_d x_d + b$$

Here, $\mathbf{w}$ is a d dimensional *weight vector* and b is a scalar, called the *bias*.

For points that lie on the hyperplane, we have

$$h(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b = 0$$

The weight vector $\mathbf{w}$ specifies the direction that is orthogonal or normal to the hyperplane, which fixes the orientation of the hyperplane, whereas the bias b fixes the offset of the hyperplane in the d-dimensional space, i.e., where the hyperplane intersects each of the axes:

$$w_i x_i = -b \quad \text{or} \quad x_i = \frac{-b}{w_i}$$

Separating Hyperplane

A hyperplane splits the d-dimensional data space into two *half-spaces*.

A dataset is said to be *linearly separable* if each half-space has points only from a single class.

If the input dataset is linearly separable, then we can find a *separating* hyperplane $h(\mathbf{x}) = 0$, such that for all points labeled $y_i = -1$, we have $h(\mathbf{x}_i) < 0$, and for all points labeled $y_i = +1$, we have $h(\mathbf{x}_i) > 0$.

The hyperplane function $h(\mathbf{x})$ thus serves as a linear classifier or a linear discriminant, which predicts the class y for any given point $\mathbf{x}$, according to the decision rule:

$$y = \begin{cases} +1 & \text{if } h(\mathbf{x}) > 0 \\ -1 & \text{if } h(\mathbf{x}) < 0 \end{cases}$$

Geometry of a Hyperplane: Distance

Consider a point $\mathbf{x} \in \mathbb{R}^d$ that does not lie on the hyperplane. Let $\mathbf{x}_p$ be the orthogonal projection of $\mathbf{x}$ on the hyperplane, and let $\mathbf{r} = \mathbf{x} - \mathbf{x}_p$. Then we can write $\mathbf{x}$ as

$$\mathbf{x} = \mathbf{x}_p + \mathbf{r} = \mathbf{x}_p + r\frac{\mathbf{w}}{\|\mathbf{w}\|}$$

where r is the *directed distance* of the point $\mathbf{x}$ from $\mathbf{x}_p$.
To obtain an expression for r, consider the value $h(\mathbf{x})$, we have:

$$h(\mathbf{x}) = h\left(\mathbf{x}_p + r\frac{\mathbf{w}}{\|\mathbf{w}\|}\right) = \mathbf{w}^T\left(\mathbf{x}_p + r\frac{\mathbf{w}}{\|\mathbf{w}\|}\right) + b = r\|\mathbf{w}\|$$

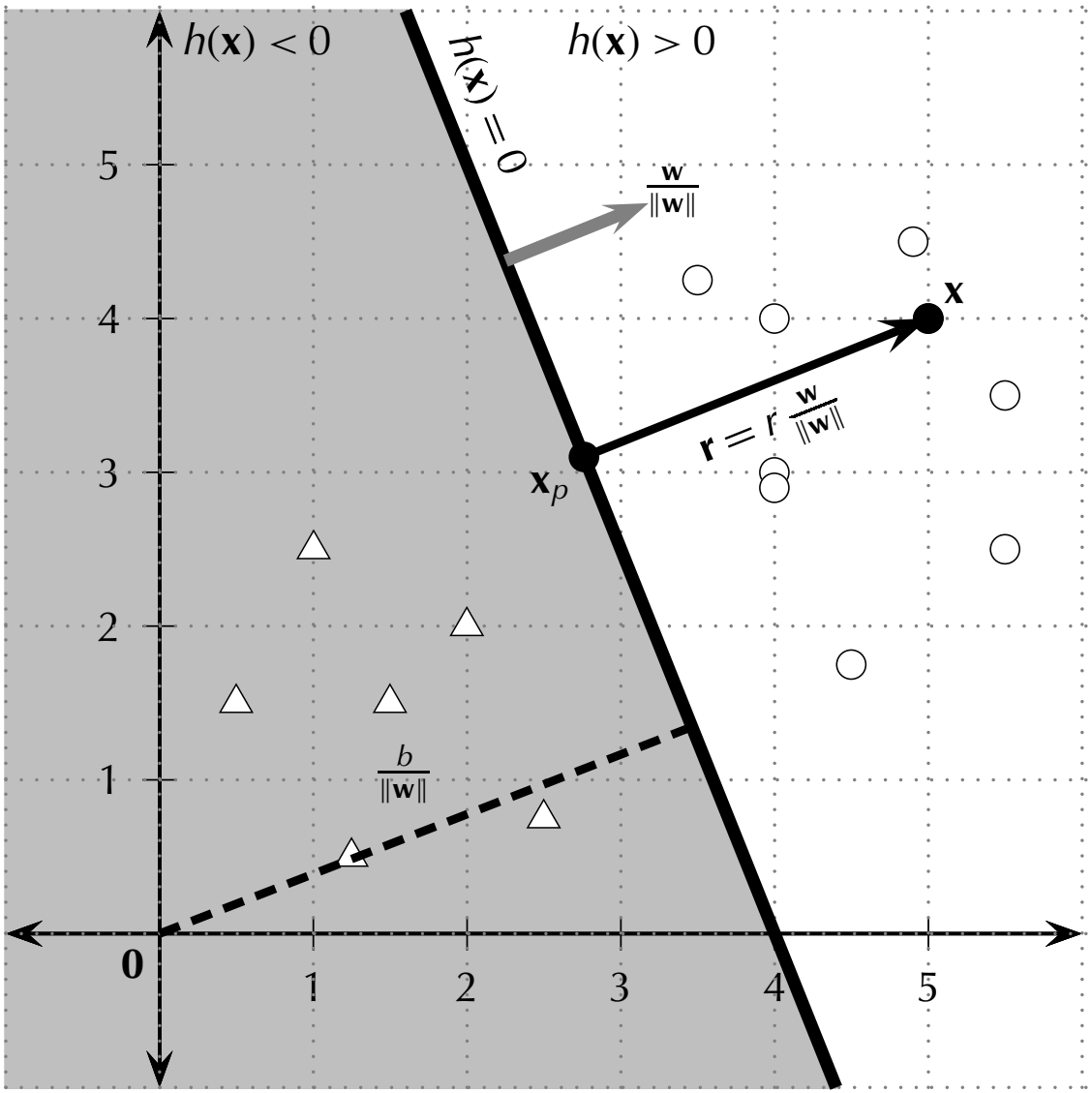
The directed distance r of point $\mathbf{x}$ to the hyperplane is thus:

$$r = \frac{h(\mathbf{x})}{\|\mathbf{w}\|}$$

To obtain distance, which must be non-negative, we multiply r by the class label y_i of the point $\mathbf{x}_i$ because when $h(\mathbf{x}_i) < 0$, the class is -1, and when $h(\mathbf{x}_i) > 0$ the class is $+1$:

$$\delta_i = \frac{y_i h(\mathbf{x}_i)}{\|\mathbf{w}\|}$$

Geometry of a Hyperplane in 2D



Margin and Support Vectors

The distance of a point $\mathbf{x}$ from the hyperplane $h(\mathbf{x}) = 0$ is thus given as

$$\delta = y\, r = \frac{y\, h(\mathbf{x})}{\|\mathbf{w}\|}$$

The *margin* is the minimum distance of a point from the separating hyperplane:

$$\delta^* = \min_{\mathbf{x}_i} \left\{ \frac{y_i(\mathbf{w}^T\mathbf{x}_i + b)}{\|\mathbf{w}\|} \right\}$$

All the points (or vectors) that achieve the minimum distance are called *support vectors* for the hyperplane. They satisfy the condition:

$$\delta^* = \frac{y^*(\mathbf{w}^T\mathbf{x}^* + b)}{\|\mathbf{w}\|}$$

where y^* is the class label for $\mathbf{x}^*$.

Canonical Hyperplane

Multiplying the hyperplane equation on both sides by some scalar s yields an equivalent hyperplane:

$$s\,h(\mathbf{x}) = s\,\mathbf{w}^T\mathbf{x} + s\,b = (s\mathbf{w})^T\mathbf{x} + (sb) = 0$$

To obtain the unique or *canonical* hyperplane, we choose the scalar $s = \frac{1}{y^*(\mathbf{w}^T\mathbf{x}^* + b)}$ so that the absolute distance of a support vector from the hyperplane is 1, i.e., the margin is

$$\delta^* = \frac{y^*(\mathbf{w}^T\mathbf{x}^* + b)}{\|\mathbf{w}\|} = \frac{1}{\|\mathbf{w}\|}$$

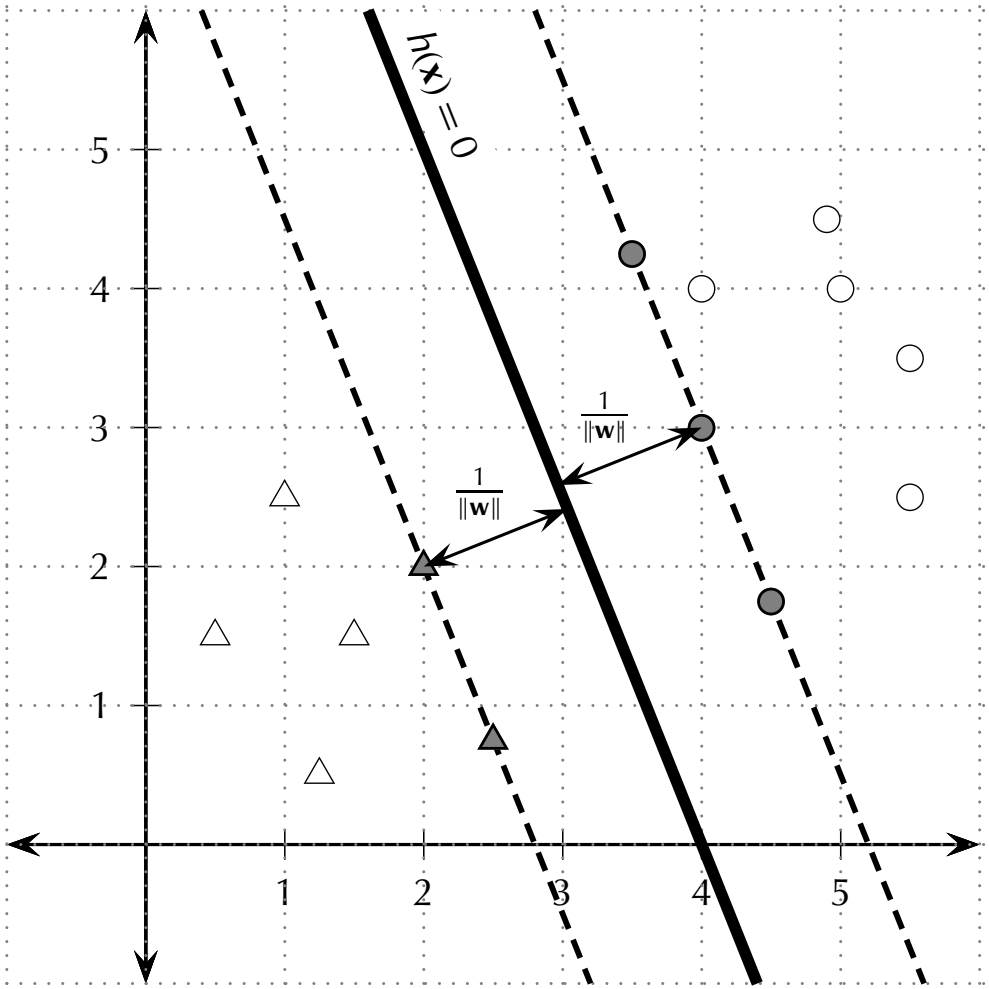
For the canonical hyperplane, for each support vector $\mathbf{x}_i^*$ (with label y_i^*), we have $y_i^* h(\mathbf{x}_i^*) = 1$, and for any point that is not a support vector we have $y_i h(\mathbf{x}_i) > 1$. Over all points, we have

$$y_i\,(\mathbf{w}^T\mathbf{x}_i + b) \geq 1, \text{ for all points } \mathbf{x}_i \in \mathbf{D}$$

Separating Hyperplane: Margin and Support Vectors

Shaded points are support vectors

Canonical hyperplane: $h(x) = 5/6x + 2/6y - 20/6 = 0.334x + 0.833y - 3.332$



SVM: Linear and Separable Case

Assume that the points are linearly separable, that is, there exists a separating hyperplane that perfectly classifies each point.

The goal of SVMs is to choose the canonical hyperplane, h^*, that yields the maximum margin among all possible separating hyperplanes

$$h^* = \arg\max_{\mathbf{w},b} \left\{ \frac{1}{\|\mathbf{w}\|} \right\}$$

We can obtain an equivalent minimization formulation:

$$\textbf{Objective Function:} \quad \min_{\mathbf{w},b} \left\{ \frac{\|\mathbf{w}\|^2}{2} \right\}$$

$$\textbf{Linear Constraints:} \quad y_i \left(\mathbf{w}^T \mathbf{x}_i + b\right) \geq 1, \quad \forall \mathbf{x}_i \in \mathbf{D}$$

SVM: Linear and Separable Case

We turn the constrained SVM optimization into an unconstrained one by introducing a Lagrange multiplier α_i for each constraint. The new objective function, called the *Lagrangian*, then becomes

$$\min\ L = \frac{1}{2}\|\mathbf{w}\|^2 - \sum_{i=1}^{n} \alpha_i \left(y_i(\mathbf{w}^T\mathbf{x}_i + b) - 1\right)$$

L should be minimized with respect to $\mathbf{w}$ and b, and it should be maximized with respect to α_i.
Taking the derivative of L with respect to $\mathbf{w}$ and b, and setting those to zero, we obtain

$$\frac{\partial}{\partial \mathbf{w}} L = \mathbf{w} - \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i = \mathbf{0} \quad \text{or} \quad \mathbf{w} = \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i$$

$$\frac{\partial}{\partial b} L = \sum_{i=1}^{n} \alpha_i y_i = 0$$

We can see that $\mathbf{w}$ can be expressed as a linear combination of the data points $\mathbf{x}_i$, with the signed Lagrange multipliers, $\alpha_i y_i$, serving as the coefficients.

Further, the sum of the signed Lagrange multipliers, $\alpha_i y_i$, must be zero.

SVM: Linear and Separable Case

Incorporating $\mathbf{w} = \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i$ and $\sum_{i=1}^{n} \alpha_i y_i = 0$ into the Lagrangian we obtain the new *dual Lagrangian* objective function, which is specified purely in terms of the Lagrange multipliers:

$$\textbf{Objective Function:} \quad \max_{\boldsymbol{\alpha}} \; L_{dual} = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j$$

$$\textbf{Linear Constraints:} \quad \alpha_i \geq 0, \; \forall i \in \mathbf{D}, \; \text{and} \; \sum_{i=1}^{n} \alpha_i y_i = 0$$

where $\boldsymbol{\alpha} = (\alpha_1, \alpha_2, \ldots, \alpha_n)^T$ is the vector comprising the Lagrange multipliers.

L_{dual} is a convex quadratic programming problem (note the $\alpha_i \alpha_j$ terms), which admits a unique optimal solution.

SVM: Linear and Separable Case

Once we have obtained the α_i values for $i = 1, \ldots, n$, we can solve for the weight vector $\mathbf{w}$ and the bias b. Each of the Lagrange multipliers α_i satisfies the KKT conditions at the optimal solution:

$$\alpha_i \left(y_i(\mathbf{w}^T \mathbf{x}_i + b) - 1 \right) = 0$$

which gives rise to two cases:

(1) $\alpha_i = 0$, or

(2) $y_i(\mathbf{w}^T \mathbf{x}_i + b) - 1 = 0$, which implies $y_i(\mathbf{w}^T \mathbf{x}_i + b) = 1$

This is a very important result because if $\alpha_i > 0$, then $y_i(\mathbf{w}^T \mathbf{x}_i + b) = 1$, and thus the point $\mathbf{x}_i$ must be a support vector.

On the other hand, if $y_i(\mathbf{w}^T \mathbf{x}_i + b) > 1$, then $\alpha_i = 0$, that is, if a point is not a support vector, then $\alpha_i = 0$.

Linear and Separable Case: Weight Vector and Bias

Once we know α_i for all points, we can compute the weight vector $\mathbf{w}$ by taking the summation only for the support vectors:

$$\mathbf{w} = \sum_{i,\alpha_i>0} \alpha_i y_i \mathbf{x}_i$$

Only the support vectors determine $\mathbf{w}$, since $\alpha_i = 0$ for other points.
To compute the bias b, we first compute one solution b_i, per support vector, as follows:

$$y_i(\mathbf{w}^T\mathbf{x}_i + b) = 1, \text{ which implies } b_i = \frac{1}{y_i} - \mathbf{w}^T\mathbf{x}_i = y_i - \mathbf{w}^T\mathbf{x}_i$$

The bias b is taken as the average value:

$$b = \text{avg}_{\alpha_i>0}\{b_i\}$$

SVM Classifier

Given the optimal hyperplane function $h(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b$, for any new point $\mathbf{z}$, we predict its class as

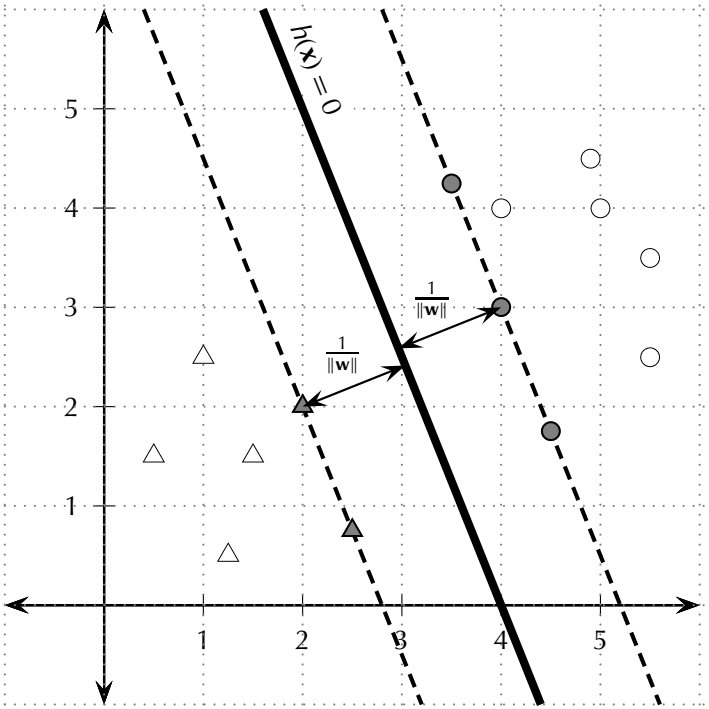
$$\hat{y} = \text{sign}(h(\mathbf{z})) = \text{sign}(\mathbf{w}^T\mathbf{z} + b)$$

where the sign$(\cdot)$ function returns $+1$ if its argument is positive, and -1 if its argument is negative.

Example Dataset: Separable Case

$\mathbf{x}_i$	x_{i1}	x_{i2}	y_i
$\mathbf{x}_1$	3.5	4.25	+1
$\mathbf{x}_2$	4	3	+1
$\mathbf{x}_3$	4	4	+1
$\mathbf{x}_4$	4.5	1.75	+1
$\mathbf{x}_5$	4.9	4.5	+1
$\mathbf{x}_6$	5	4	+1
$\mathbf{x}_7$	5.5	2.5	+1
$\mathbf{x}_8$	5.5	3.5	+1
$\mathbf{x}_9$	0.5	1.5	−1
$\mathbf{x}_{10}$	1	2.5	−1
$\mathbf{x}_{11}$	1.25	0.5	−1
$\mathbf{x}_{12}$	1.5	1.5	−1
$\mathbf{x}_{13}$	2	2	−1
$\mathbf{x}_{14}$	2.5	0.75	−1

Optimal Separating Hyperplane



Solving the L_{dual} quadratic program yields

$\mathbf{x}_i$	x_{i1}	x_{i2}	y_i	α_i
$\mathbf{x}_1$	3.5	4.25	+1	0.0437
$\mathbf{x}_2$	4	3	+1	0.2162
$\mathbf{x}_4$	4.5	1.75	+1	0.1427
$\mathbf{x}_{13}$	2	2	−1	0.3589
$\mathbf{x}_{14}$	2.5	0.75	−1	0.0437

The weight vector and bias are:

$$\mathbf{w} = \sum_{i,\alpha_i>0} \alpha_i y_i \mathbf{x}_i = \begin{pmatrix} 0.833 \\ 0.334 \end{pmatrix}$$

$$b = \text{avg}\{b_i\} = -3.332$$

The optimal hyperplane is given as follows:

$$h(\mathbf{x}) = \begin{pmatrix} 0.833 \\ 0.334 \end{pmatrix}^T \mathbf{x} - 3.332 = 0$$

Soft Margin SVM: Linear and Nonseparable Case

The assumption that the dataset be perfectly linearly separable is unrealistic. SVMs can handle non-separable points by introducing *slack variables* ξ_i as follows:

$$y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1 - \xi_i$$

where $\xi_i \geq 0$ is the slack variable for point $\mathbf{x}_i$, which indicates how much the point violates the separability condition, that is, the point may no longer be at least $1/\|\mathbf{w}\|$ away from the hyperplane.

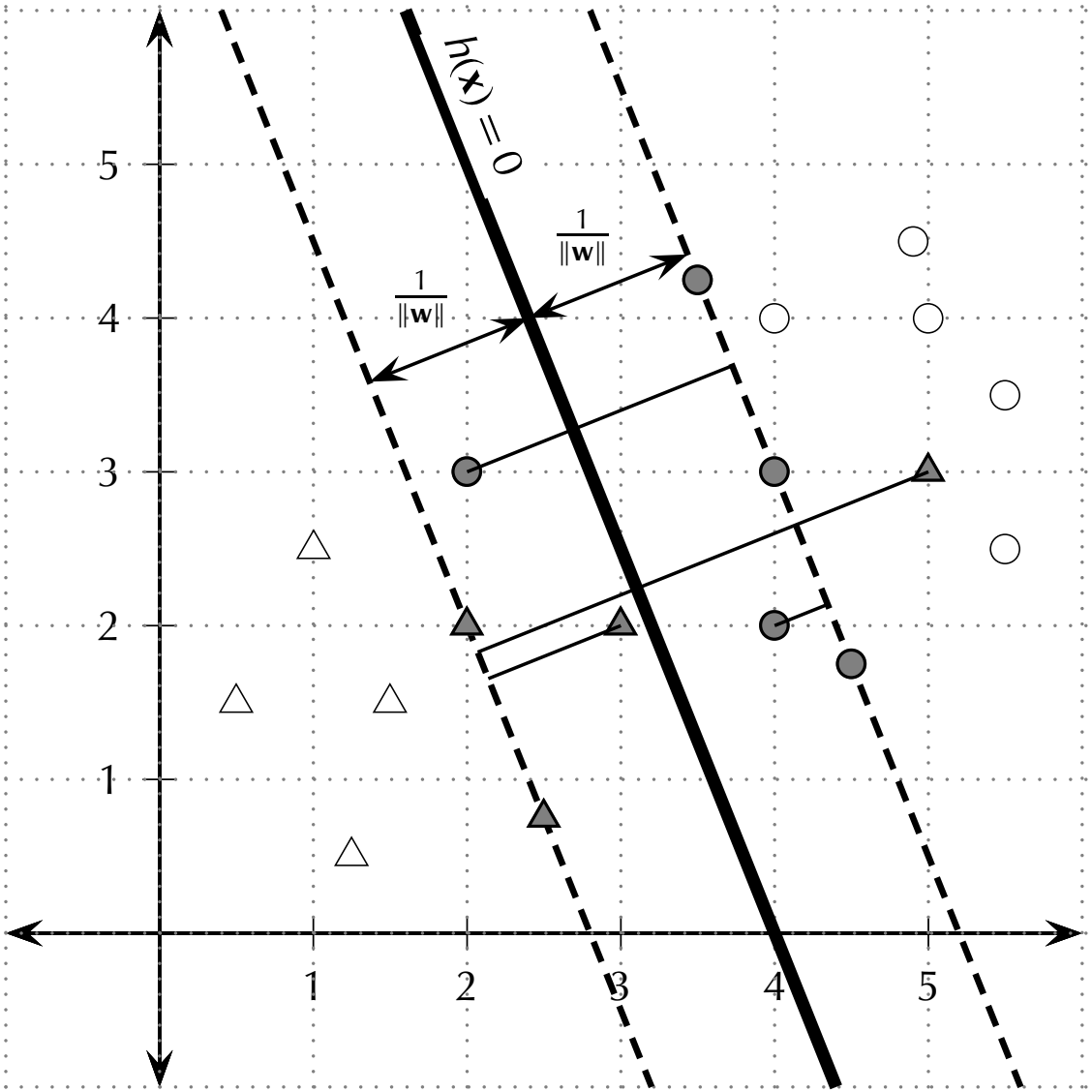
The slack values indicate three types of points. If $\xi_i = 0$, then the corresponding point $\mathbf{x}_i$ is at least $\frac{1}{\|\mathbf{w}\|}$ away from the hyperplane.

If $0 < \xi_i < 1$, then the point is within the margin and still correctly classified, that is, it is on the correct side of the hyperplane.

However, if $\xi_i \geq 1$ then the point is misclassified and appears on the wrong side of the hyperplane.

Soft Margin Hyperplane

Shaded points are the support vectors



SVM: Soft Margin or Linearly Non-separable Case

In the nonseparable case, also called the *soft margin* the SVM objective function is

$$\textbf{Objective Function:} \quad \min_{\mathbf{w},b,\xi_i} \left\{ \frac{\|\mathbf{w}\|^2}{2} + C\sum_{i=1}^{n} (\xi_i)^k \right\}$$

$$\textbf{Linear Constraints:} \quad y_i \left(\mathbf{w}^T\mathbf{x}_i + b\right) \geq 1 - \xi_i, \quad \forall \mathbf{x}_i \in \mathbf{D}$$

$$\xi_i \geq 0 \quad \forall \mathbf{x}_i \in \mathbf{D}$$

where C and k are constants that incorporate the cost of misclassification.

The term $\sum_{i=1}^{n} (\xi_i)^k$ gives the *loss*, that is, an estimate of the deviation from the separable case.

The scalar C is a *regularization constant* that controls the trade-off between maximizing the margin or minimizing the loss. For example, if $C \to 0$, then the loss component essentially disappears, and the objective defaults to maximizing the margin. On the other hand, if $C \to \infty$, then the margin ceases to have much effect, and the objective function tries to minimize the loss.

SVM: Soft Margin Loss Function

The constant k governs the form of the loss. When $k=1$, called *hinge loss*, the goal is to minimize the sum of the slack variables, whereas when $k=2$, called *quadratic loss*, the goal is to minimize the sum of the squared slack variables.

Hinge Loss: Assuming $k=1$, the SVM dual Lagrangian is given as

$$\max_{\boldsymbol{\alpha}} \; L_{dual} = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j$$

The only difference from the separable case is that $0 \leq \alpha_i \leq C$.

Quadratic Loss: Assuming $k=2$, the dual objective is:

$$\max_{\boldsymbol{\alpha}} \; L_{dual} = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \left(\mathbf{x}_i^T \mathbf{x}_j + \frac{1}{2C} \delta_{ij} \right)$$

where δ is the *Kronecker delta* function, defined as $\delta_{ij} = 1$ if and only if $i = j$.

Example Dataset: Linearly Non-separable Case

$\mathbf{x}_i$	x_{i1}	x_{i2}	y_i
$\mathbf{x}_1$	3.5	4.25	+1
$\mathbf{x}_2$	4	3	+1
$\mathbf{x}_3$	4	4	+1
$\mathbf{x}_4$	4.5	1.75	+1
$\mathbf{x}_5$	4.9	4.5	+1
$\mathbf{x}_6$	5	4	+1
$\mathbf{x}_7$	5.5	2.5	+1
$\mathbf{x}_8$	5.5	3.5	+1
$\mathbf{x}_9$	0.5	1.5	−1
$\mathbf{x}_{10}$	1	2.5	−1
$\mathbf{x}_{11}$	1.25	0.5	−1
$\mathbf{x}_{12}$	1.5	1.5	−1
$\mathbf{x}_{13}$	2	2	−1
$\mathbf{x}_{14}$	2.5	0.75	−1
$\mathbf{x}_{15}$	4	2	+1
$\mathbf{x}_{16}$	2	3	+1
$\mathbf{x}_{17}$	3	2	−1
$\mathbf{x}_{18}$	5	3	−1

Example Dataset: Linearly Non-separable Case

Let $k = 1$ and $C = 1$, then solving the L_{dual} yields the following support vectors and Lagrangian values α_i:

$\mathbf{x}_i$	x_{i1}	x_{i2}	y_i	α_i
$\mathbf{x}_1$	3.5	4.25	+1	0.0271
$\mathbf{x}_2$	4	3	+1	0.2162
$\mathbf{x}_4$	4.5	1.75	+1	0.9928
$\mathbf{x}_{13}$	2	2	−1	0.9928
$\mathbf{x}_{14}$	2.5	0.75	−1	0.2434
$\mathbf{x}_{15}$	4	2	+1	1
$\mathbf{x}_{16}$	2	3	+1	1
$\mathbf{x}_{17}$	3	2	−1	1
$\mathbf{x}_{18}$	5	3	−1	1

The optimal hyperplane is given as follows:

$$h(\mathbf{x}) = \begin{pmatrix} 0.834 \\ 0.333 \end{pmatrix}^T \mathbf{x} - 3.334 = 0$$

Example Dataset: Linearly Non-separable Case

The slack $\xi_i = 0$ for all points that are not support vectors, and also for those support vectors that are on the margin. Slack is positive only for the remaining support vectors and it can be computed as: $\xi_i = 1 - y_i(\mathbf{w}^T\mathbf{x}_i + b)$.
Thus, for all support vectors not on the margin, we have

$\mathbf{x}_i$	$\mathbf{w}^T\mathbf{x}_i$	$\mathbf{w}^T\mathbf{x}_i + b$	$\xi_i = 1 - y_i(\mathbf{w}^T\mathbf{x}_i + b)$
$\mathbf{x}_{15}$	4.001	0.667	0.333
$\mathbf{x}_{16}$	2.667	−0.667	1.667
$\mathbf{x}_{17}$	3.167	−0.167	0.833
$\mathbf{x}_{18}$	5.168	1.834	2.834

The total slack is given as

$$\sum_i \xi_i = \xi_{15} + \xi_{16} + \xi_{17} + \xi_{18} = 0.333 + 1.667 + 0.833 + 2.834 = 5.667$$

The slack variable $\xi_i > 1$ for those points that are misclassified (i.e., are on the wrong side of the hyperplane), namely $\mathbf{x}_{16} = (3, 3)^T$ and $\mathbf{x}_{18} = (5, 3)^T$. The other two points are correctly classified, but lie within the margin, and thus satisfy $0 < \xi_i < 1$.

Kernel SVM: Nonlinear Case

The linear SVM approach can be used for datasets with a nonlinear decision boundary via the kernel trick.

Conceptually, the idea is to map the original d-dimensional points $\mathbf{x}_i$ in the input space to points $\phi(\mathbf{x}_i)$ in a high-dimensional feature space via some nonlinear transformation ϕ.

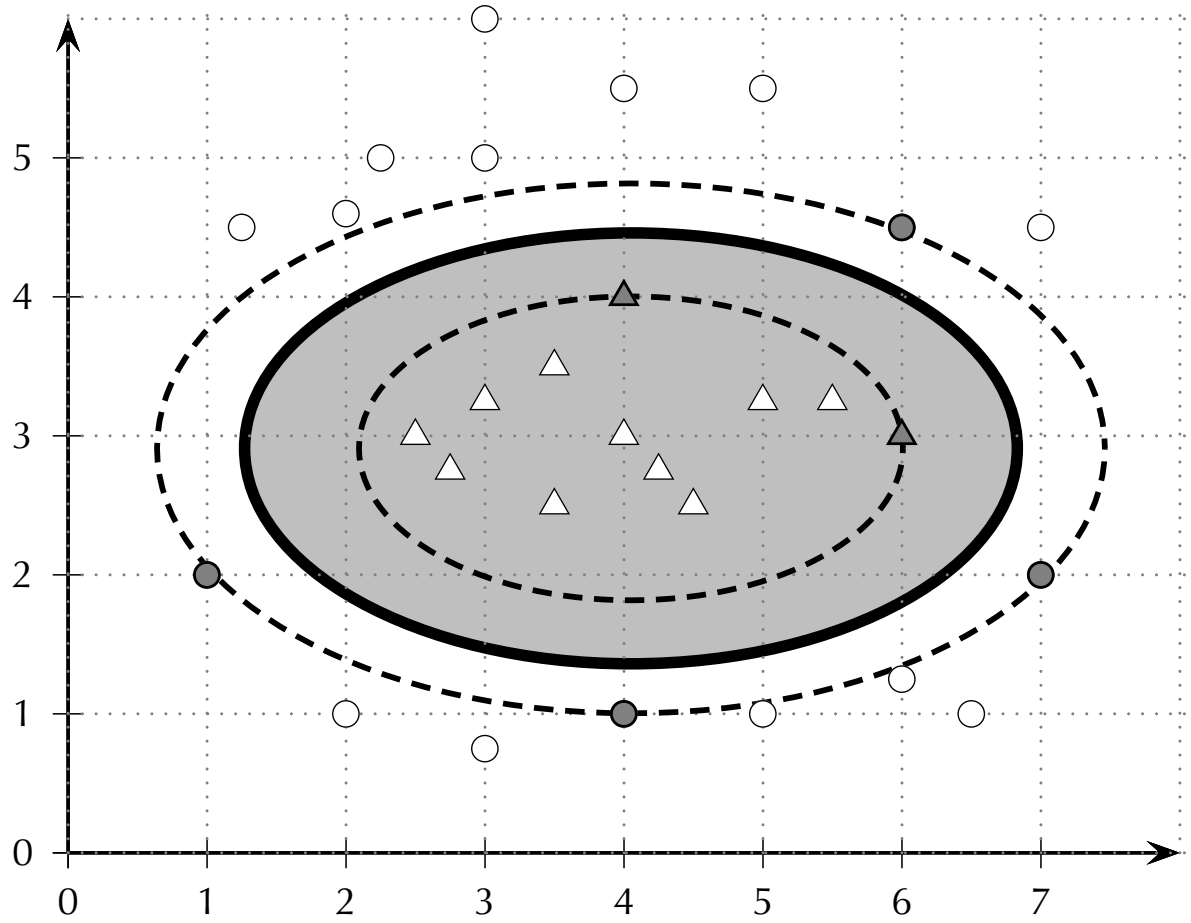
Given the extra flexibility, it is more likely that the points $\phi(\mathbf{x}_i)$ might be linearly separable in the feature space.

A linear decision surface in feature space actually corresponds to a nonlinear decision surface in the input space.

Further, the kernel trick allows us to carry out all operations via the kernel function computed in input space, rather than having to map the points into feature space.

Nonlinear SVM

There is no linear classifier that can discriminate between the points. However, there exists a perfect quadratic classifier that can separate the two classes.



Paradigms

- Combinatorial
- Probabilistic
- Algebraic
- **Graph-based**

Domain

Input data is modeled as a graph, enabling not just richer representations but also several existing models and algorithms.

Task

Determine the best representation and technique, according to an optimization metric.

Challenge

How can we handle the larger complexity and numerosity induced by graphs?

Graphs

A *graph* $G = (V, E)$ comprises a finite nonempty set V of *vertices* or *nodes*, and a set $E \subseteq V \times V$ of *edges* consisting of *unordered* pairs of vertices.

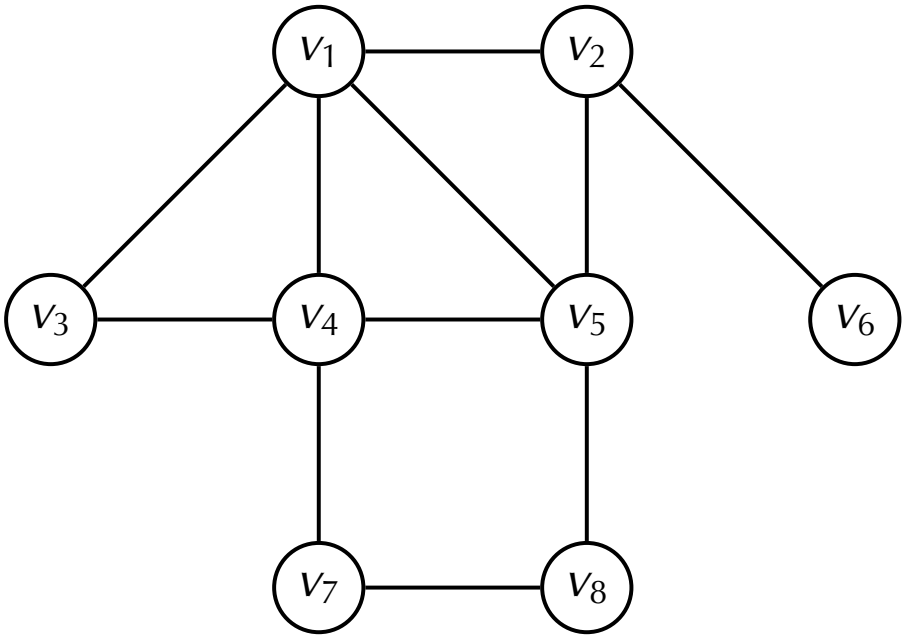
The number of nodes in the graph G, given as $|V| = n$, is called the *order* of the graph, and the number of edges in the graph, given as $|E| = m$, is called the *size* of G.

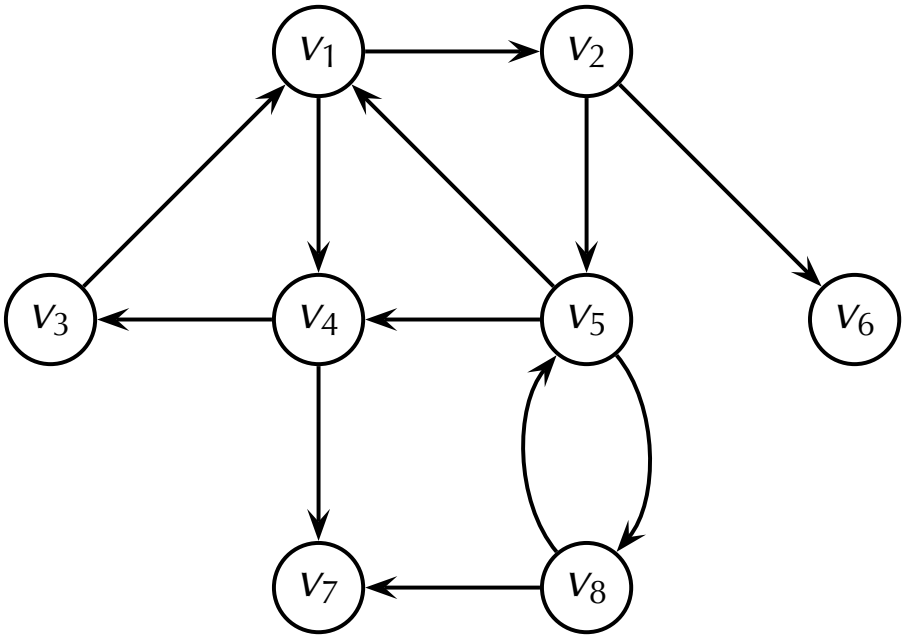
A *directed graph* or *digraph* has an edge set E consisting of *ordered* pairs of vertices.

A *weighted graph* consists of a graph together with a weight w_{ij} for each edge $(v_i, v_j) \in E$.

A graph $H = (V_H, E_H)$ is called a *subgraph* of $G = (V, E)$ if $V_H \subseteq V$ and $E_H \subseteq E$.

Undirected and Directed Graphs





Degree Distribution

The *degree* of a node $v_i \in V$ is the number of edges incident with it, and is denoted as $d(v_i)$ or just d_i.

The *degree sequence* of a graph is the list of the degrees of the nodes sorted in non-increasing order.

Let N_k denote the number of vertices with degree k. The *degree frequency distribution* of a graph is given as

$$(N_0, N_1, \ldots, N_t)$$

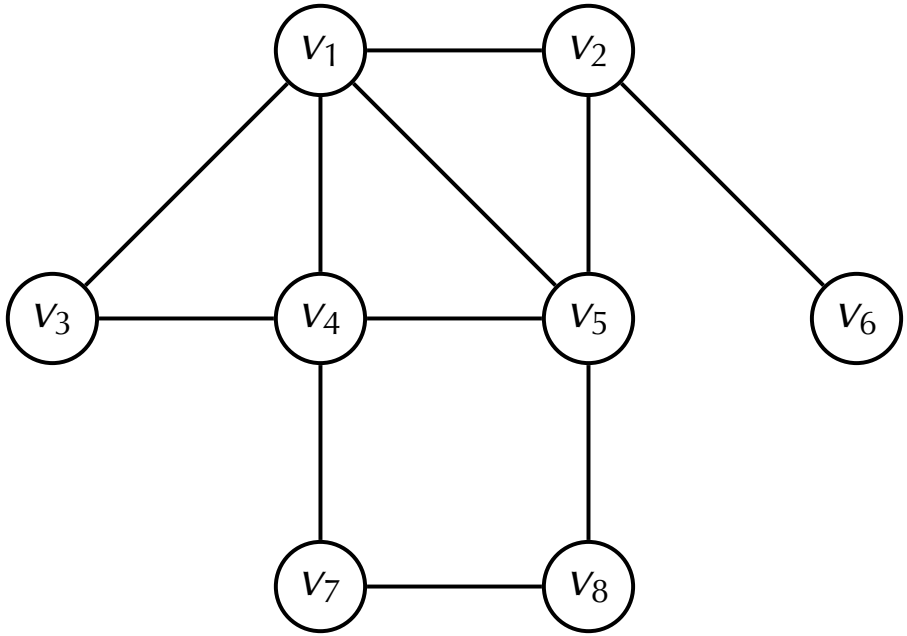
where t is the maximum degree for a node in G.

Let X be a random variable denoting the degree of a node. The *degree distribution* of a graph gives the probability mass function f for X, given as

$$\big(f(0), f(1), \ldots, f(t)\big)$$

where $f(k) = P(X = k) = \frac{N_k}{n}$ is the probability of a node with degree k.

Degree Distribution



The degree sequence of the graph is

$$(4, 4, 4, 3, 2, 2, 2, 1)$$

Its degree frequency distribution is

$$(N_0, N_1, N_2, N_3, N_4) = (0, 1, 3, 1, 3)$$

The degree distribution is given as

$$\big(f(0), f(1), f(2), f(3), f(4)\big) = (0, 0.125, 0.375, 0.125, 0.375)$$

- **Frequent Subgraph Mining**
- Spectral Clustering

Unlabeled and Labeled Graphs

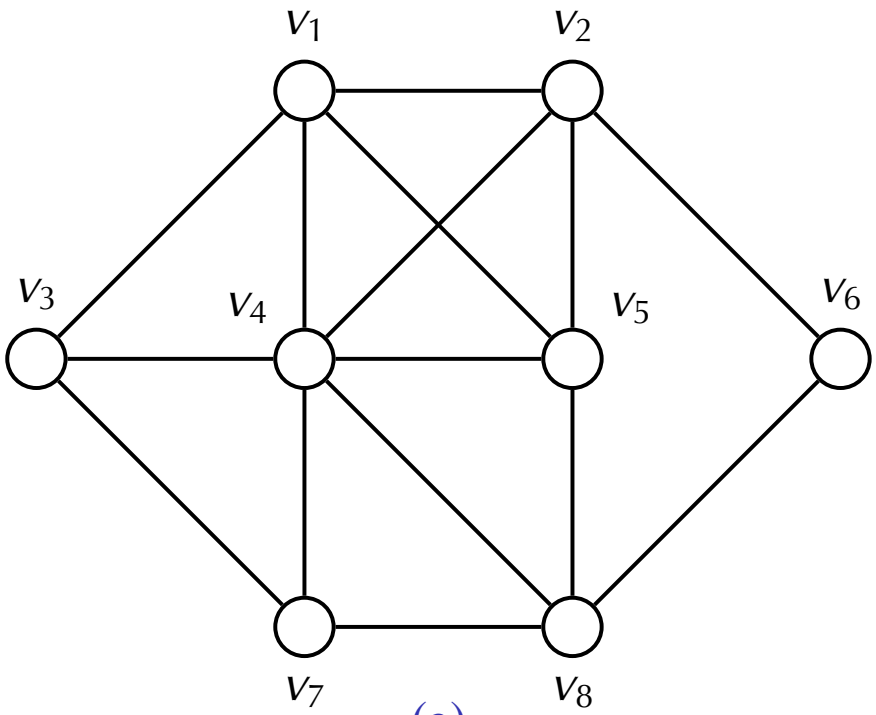
The goal of graph mining is to extract interesting subgraphs from a single large graph (e.g., a social network), or from a database of many graphs.

A graph is a pair $G = (V, E)$ where V is a set of vertices, and $E \subseteq V \times V$ is a set of edges.

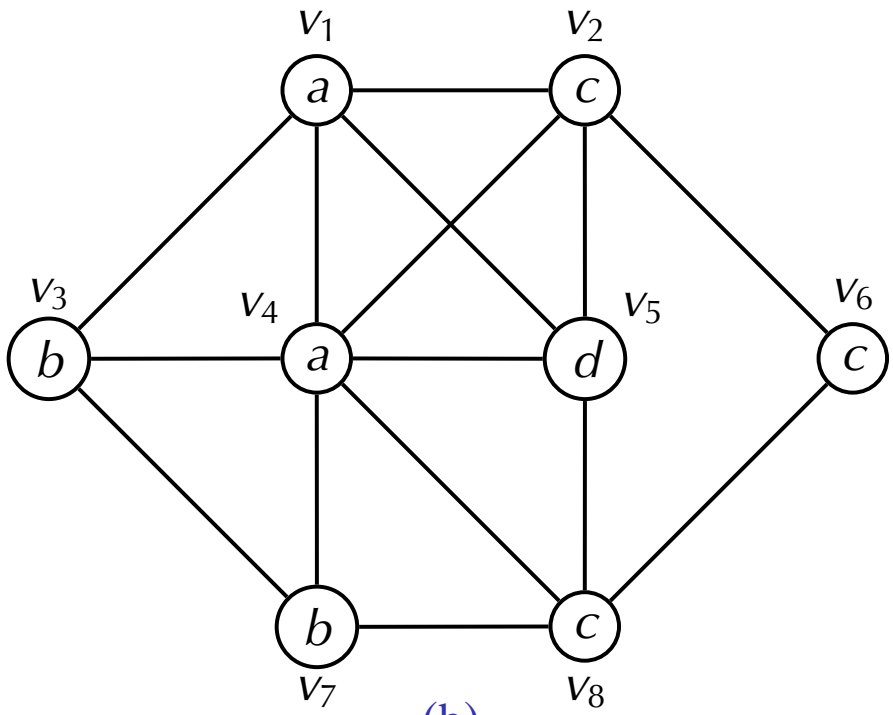
A *labeled graph* has labels associated with its vertices as well as edges. We use $L(u)$ to denote the label of the vertex u, and $L(u, v)$ to denote the label of the edge (u, v), with the set of vertex labels denoted as Σ_V and the set of edge labels as Σ_E. Given an edge $(u, v) \in G$, the tuple $\langle u, v, L(u), L(v), L(u, v)\rangle$ that augments the edge with the node and edge labels is called an *extended edge*.

A graph $G' = (V', E')$ is said to be a *subgraph* of G if $V' \subseteq V$ and $E' \subseteq E$. A *connected subgraph* is defined as a subgraph G' such that $V' \subseteq V$, $E' \subseteq E$, and for any two nodes $u, v \in V'$, there exists a *path* from u to v in G'.

Unlabeled and Labeled Graphs

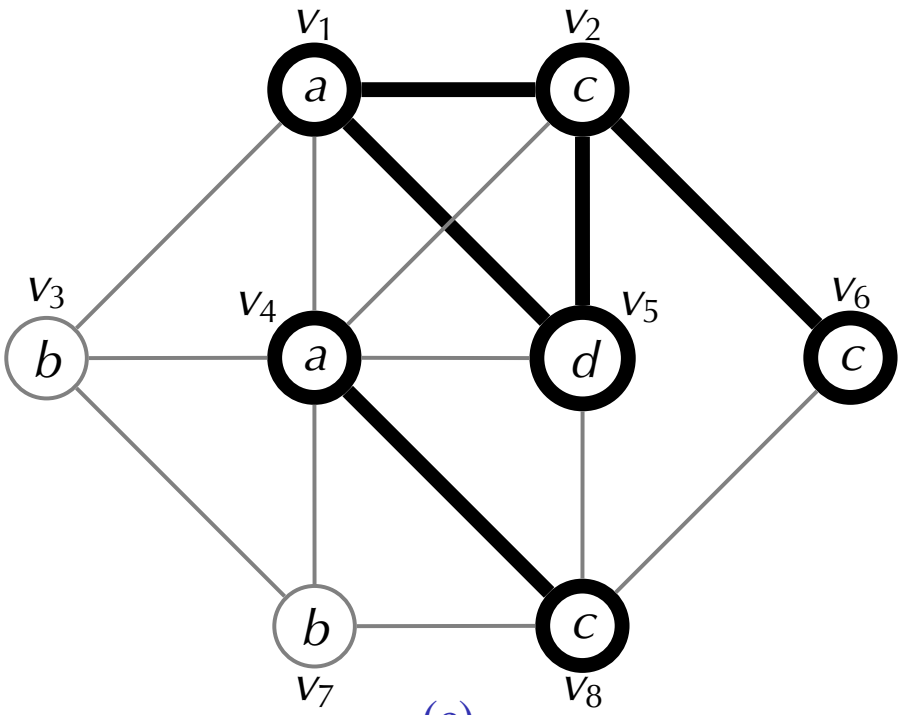


(a)

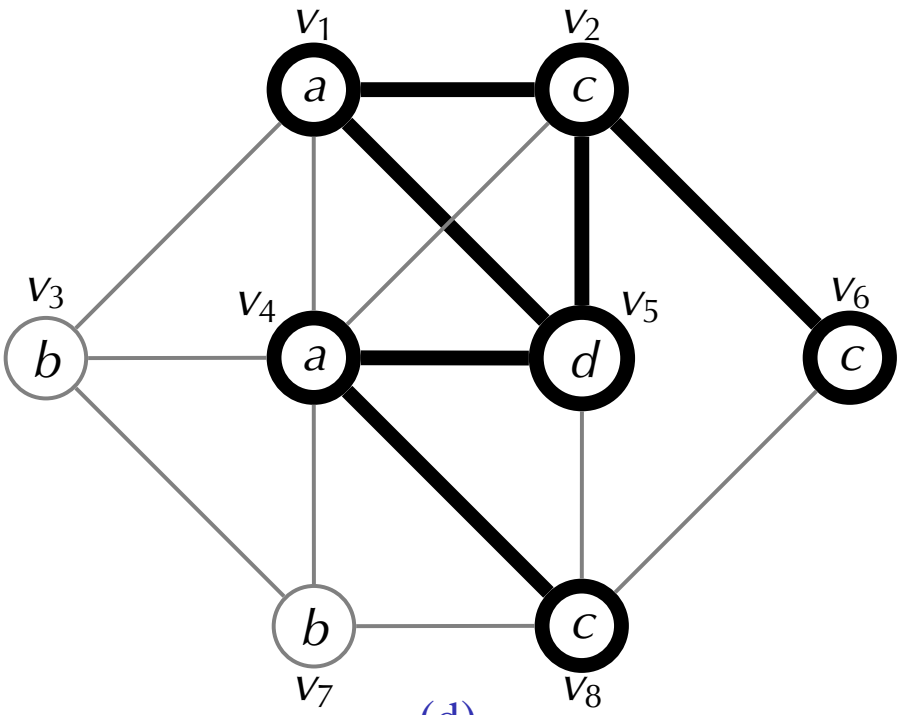


(b)

Subgraph and Connected Subgraph



(c)



(d)

Graph and Subgraph Isomorphism

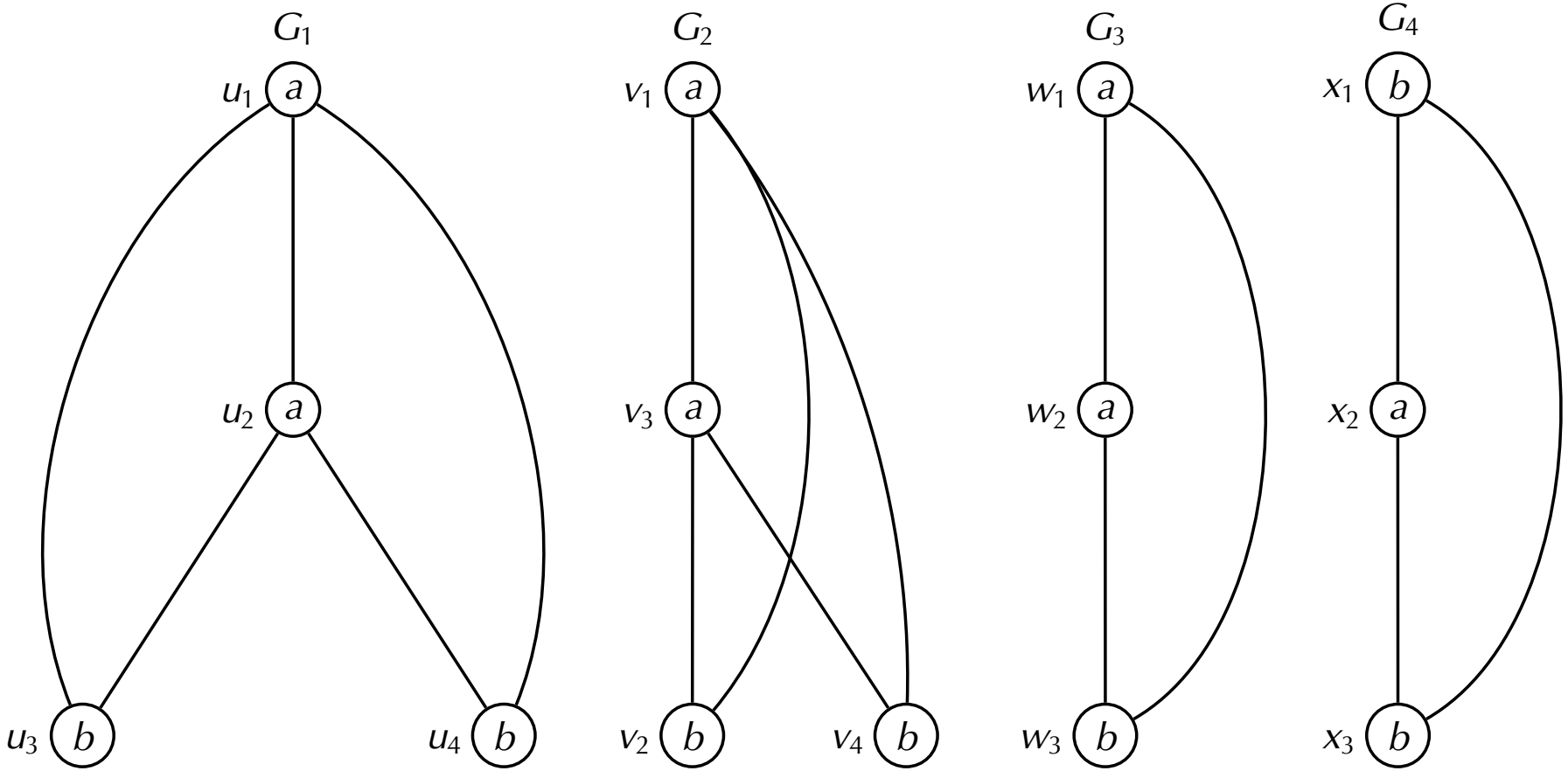
A graph $G' = (V', E')$ is said to be *isomorphic* to another graph $G = (V, E)$ if there exists a bijective function $\phi : V' \rightarrow V$, i.e., both injective (into) and surjective (onto), such that

- $(u, v) \in E' \iff (\phi(u), \phi(v)) \in E$
- $\forall u \in V',\ L(u) = L(\phi(u))$
- $\forall (u, v) \in E',\ L(u, v) = L(\phi(u), \phi(v))$

In other words, the *isomorphism* ϕ preserves the edge adjacencies as well as the vertex and edge labels. Put differently, the extended tuple $\langle u, v, L(u), L(v), L(u, v)\rangle \in G'$ if and only if $\langle \phi(u), \phi(v), L(\phi(u)), L(\phi(v)), L(\phi(u), \phi(v))\rangle \in G$.

If the function ϕ is only injective but not surjective, we say that the mapping ϕ is a *subgraph isomorphism* from G' to G. In this case, we say that G' is isomorphic to a subgraph of G, that is, G' is *subgraph isomorphic* to G, denoted $G' \subseteq G$; we also say that G *contains* G'.

Graph and Subgraph Isomorphism



Graph Isomorphism

G_1 and G_2 are isomorphic graphs. There are several possible isomorphisms between G_1 and G_2. An example of an isomorphism $\phi : V_2 \rightarrow V_1$ is

$$\phi(v_1) = u_1 \qquad \phi(v_2) = u_3 \qquad \phi(v_3) = u_2 \qquad \phi(v_4) = u_4$$

The inverse mapping ϕ^{-1} specifies the isomorphism from G_1 to G_2. For example, $\phi^{-1}(u_1) = v_1$, $\phi^{-1}(u_2) = v_3$, and so on.

The set of all possible isomorphisms from G_2 to G_1 are as follows:

	v_1	v_2	v_3	v_4
ϕ_1	u_1	u_3	u_2	u_4
ϕ_2	u_1	u_4	u_2	u_3
ϕ_3	u_2	u_3	u_1	u_4
ϕ_4	u_2	u_4	u_1	u_3

Subgraph Isomorphism

The graph G_3 is subgraph isomorphic to both G_1 and G_2. The set of all possible subgraph isomorphisms from G_3 to G_1 are as follows:

	w_1	w_2	w_3
ϕ_1	u_1	u_2	u_3
ϕ_2	u_1	u_2	u_4
ϕ_3	u_2	u_1	u_3
ϕ_4	u_2	u_1	u_4

Mining Frequent Subgraph

Given a database of graphs, $\mathbf{D} = \{G_1, G_2, \ldots, G_n\}$, and given some graph G, the support of G in $\mathbf{D}$ is defined as follows:

$$sup(G) = \left| \{ G_i \in \mathbf{D} \mid G \subseteq G_i \} \right|$$

The support is simply the number of graphs in the database that contain G. Given a *minsup* threshold, the goal of graph mining is to mine all frequent connected subgraphs with $sup(G) \geq minsup$.

If we consider subgraphs with m vertices, then there are $\binom{m}{2} = O(m^2)$ possible edges. The number of possible subgraphs with m nodes is then $O(2^{m^2})$ because we may decide either to include or exclude each of the edges. Many of these subgraphs will not be connected, but $O(2^{m^2})$ is a convenient upper bound. When we add labels to the vertices and edges, the number of labeled graphs will be even more.

Graph Pattern Mining

There are two main challenges in frequent subgraph mining.

The first is to systematically generate nonredundant candidate subgraphs. We use *edge-growth* as the basic mechanism for extending the candidates.

The second challenge is to count the support of a graph in the database. This involves subgraph isomorphism checking, as we have to find the set of graphs that contain a given candidate.

Candidate Generation

An effective strategy to enumerate subgraph patterns is *rightmost path extension*.

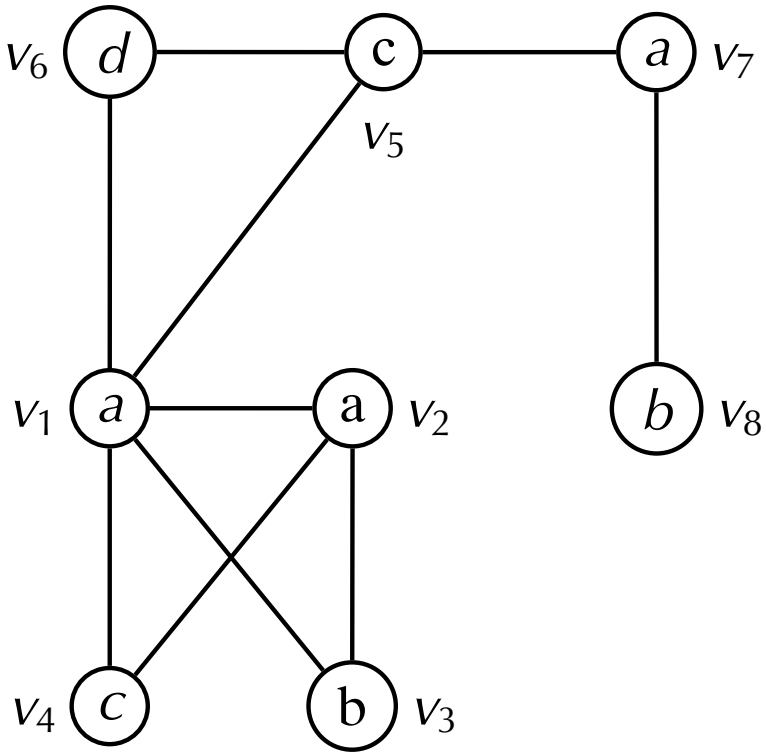
Given a graph G, we perform a depth-first search (DFS) over its vertices, and create a DFS spanning tree, that is, one that covers or spans all the vertices.

Edges that are included in the DFS tree are called *forward* edges, and all other edges are called *backward* edges. Backward edges create cycles in the graph.

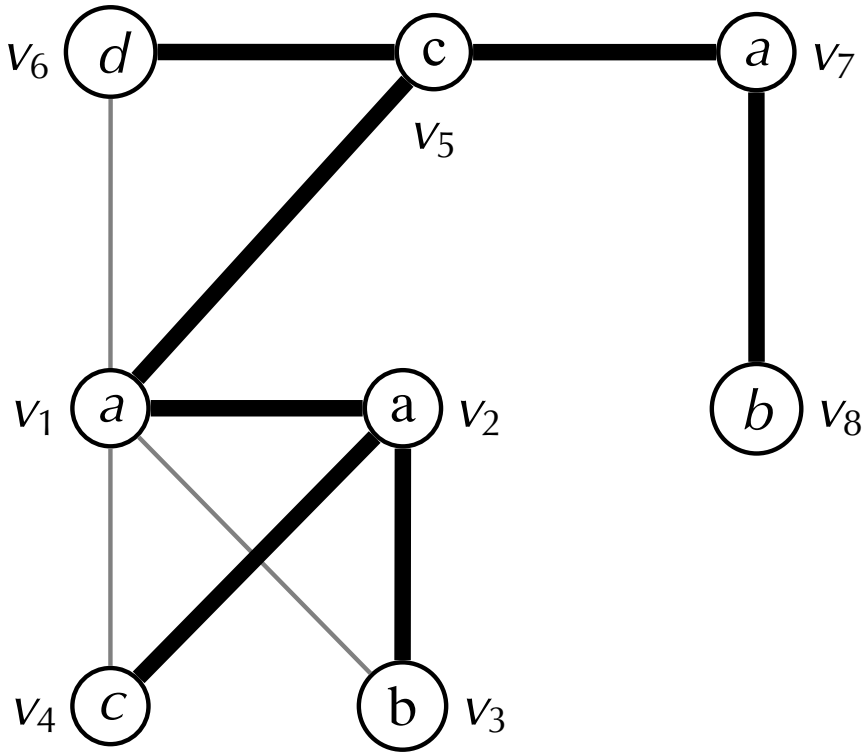
Once we have a DFS tree, define the *rightmost* path as the path from the root to the rightmost leaf, that is, to the leaf with the highest index in the DFS order.

Depth-first Spanning Tree

Starting at v_1, each DFS step chooses the vertex with smallest index



(e)



(f)

Rightmost Path Extensions

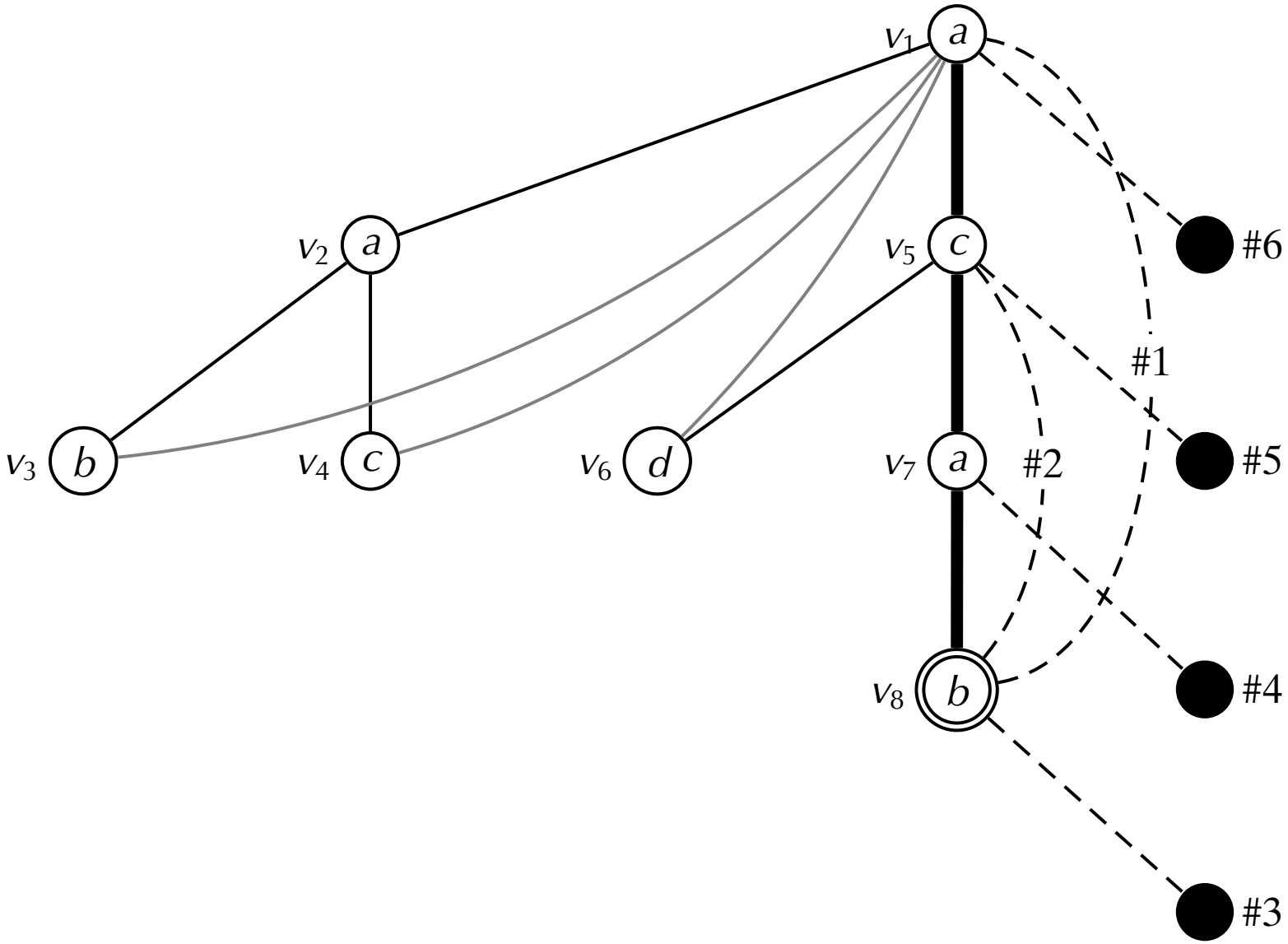
For generating new candidates from a given graph G, we extend it by adding a new edge to vertices only on the rightmost path. We can either extend G by adding backward edges from the *rightmost vertex* to some other vertex on the rightmost path (disallowing self-loops or multi-edges), or we can extend G by adding forward edges from any of the vertices on the rightmost path. A backward extension does not add a new vertex, whereas a forward extension adds a new vertex.

For systematic candidate generation we impose a total order on the extensions, as follows: First, we try all backward extensions from the rightmost vertex, and then we try forward extensions from vertices on the rightmost path.

Among the backward edge extensions, if u_r is the rightmost vertex, the extension (u_r, v_i) is tried before (u_r, v_j) if $i < j$. In other words, backward extensions closer to the root are considered before those farther away from the root along the rightmost path.

Among the forward edge extensions, if v_x is the new vertex to be added, the extension (v_i, v_x) is tried before (v_j, v_x) if $i > j$. In other words, the vertices farther from the root (those at greater depth) are extended before those closer to the root. Also note that the new vertex will be numbered $x = r + 1$, as it will become the new rightmost vertex after the extension.

Rightmost Path Extensions



DFS Code

For systematic enumeration we rank the set of isomorphic graphs and pick one member as the *canonical* representative.

Let G be a graph and let T_G be a DFS spanning tree for G. The DFS tree T_G defines an ordering of both the nodes and edges in G. The DFS node ordering is obtained by numbering the nodes consecutively in the order they are visited in the DFS walk.

Assume that for a pattern graph G the nodes are numbered according to their position in the DFS ordering, so that $i < j$ implies that v_i comes before v_j in the DFS walk.

The DFS edge ordering is obtained by following the edges between consecutive nodes in DFS order, with the condition that all the backward edges incident with vertex v_i are listed before any of the forward edges incident with it.

The *DFS code* for a graph G, for a given DFS tree T_G, denoted DFScode(G), is defined as the sequence of extended edge tuples of the form $\langle v_i, v_j, L(v_i), L(v_j), L(v_i, v_j)\rangle$ listed in the DFS edge order.

Canonical DFS Code

A subgraph is *canonical* if it has the smallest DFS code among all possible isomorphic graphs.

Let t_1 and t_2 be any two DFS code tuples:

$$t_1 = \langle v_i, v_j, L(v_i), L(v_j), L(v_i, v_j) \rangle$$
$$t_2 = \langle v_x, v_y, L(v_x), L(v_y), L(v_x, v_y) \rangle$$

We say that t_1 is smaller than t_2, written $t_1 < t_2$, iff

i) $(v_i, v_j) <_e (v_x, v_y)$, or

ii) $(v_i, v_j) = (v_x, v_y)$ and $\langle L(v_i), L(v_j), L(v_i, v_j) \rangle <_l \langle L(v_x), L(v_y), L(v_x, v_y) \rangle$

where $<_e$ is an ordering on the edges and $<_l$ is an ordering on the vertex and edge labels.

The *label order* $<_l$ is the standard lexicographic order on the vertex and edge labels.

The *edge order* $<_e$ is derived from the rules for rightmost path extension, namely that all of a node's backward extensions must be considered before any forward edge from that node, and deep DFS trees are preferred over bushy DFS trees.

Canonical DFS Code: Edge Ordering

Let $e_{ij} = (v_i, v_j)$ and $e_{xy} = (v_x, v_y)$ be any two edges. We say that $e_{ij} <_e e_{xy}$ iff

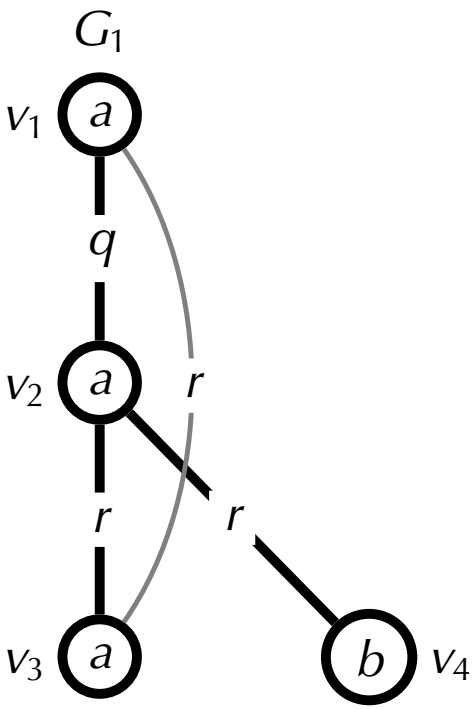
- If e_{ij} and e_{xy} are both forward edges, then (a) $j < y$, or (b) $j = y$ and $i > x$.
- If e_{ij} and e_{xy} are both backward edges, then (a) $i < x$, or (b) $i = x$ and $j < y$.
- If e_{ij} is a forward and e_{xy} is a backward edge, then $j \leq x$.
- If e_{ij} is a backward and e_{xy} is a forward edge, then $i < y$.

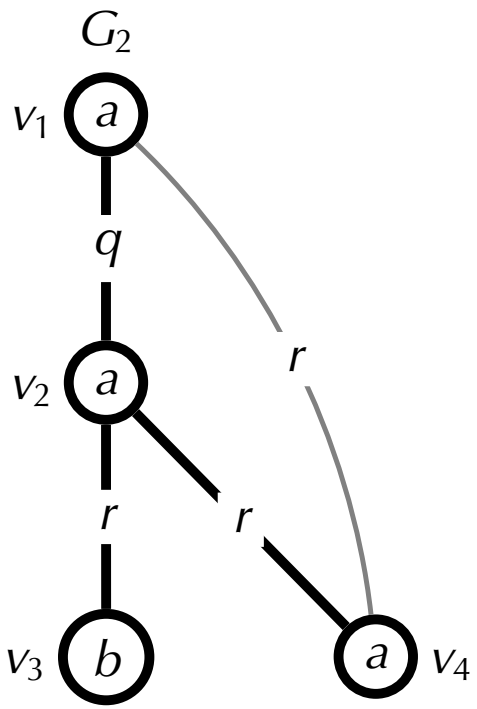
The *canonical DFS code* for a graph G is defined as follows:

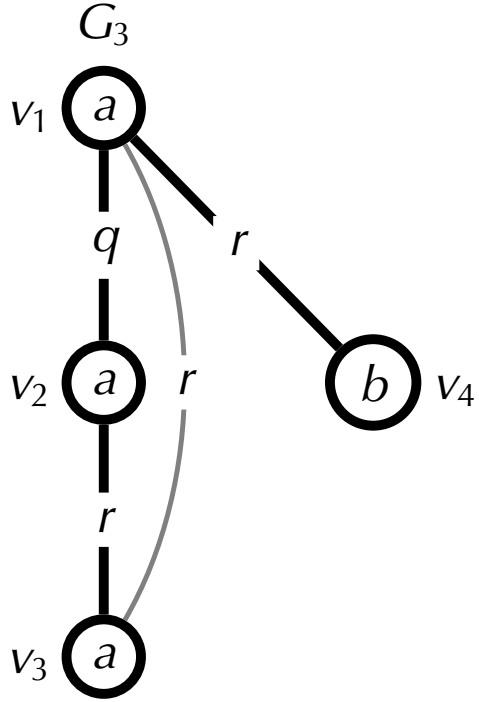
$$\mathcal{C} = \min_{G'} \left\{ \text{DFScode}(G') \mid G' \text{ is isomorphic to } G \right\}$$

Canonical DFS Code

G_1 has the canonical or minimal DFS code







$$t_{11} = \langle v_1, v_2, a, a, q \rangle$$
$$t_{12} = \langle v_2, v_3, a, a, r \rangle$$
$$t_{13} = \langle v_3, v_1, a, a, r \rangle$$
$$t_{14} = \langle v_2, v_4, a, b, r \rangle$$

DFScode(G_1)

$$t_{21} = \langle v_1, v_2, a, a, q \rangle$$
$$t_{22} = \langle v_2, v_3, a, b, r \rangle$$
$$t_{23} = \langle v_2, v_4, a, a, r \rangle$$
$$t_{24} = \langle v_4, v_1, a, a, r \rangle$$

DFScode(G_2)

$$t_{31} = \langle v_1, v_2, a, a, q \rangle$$
$$t_{32} = \langle v_2, v_3, a, a, r \rangle$$
$$t_{33} = \langle v_3, v_1, a, a, r \rangle$$
$$t_{34} = \langle v_1, v_4, a, b, r \rangle$$

DFScode(G_3)

gSpan Graph Mining Algorithm

gSpan enumerates patterns in a depth-first manner, starting with the empty code. Given a canonical and frequent code C, gSpan first determines the set of possible edge extensions along the rightmost path.

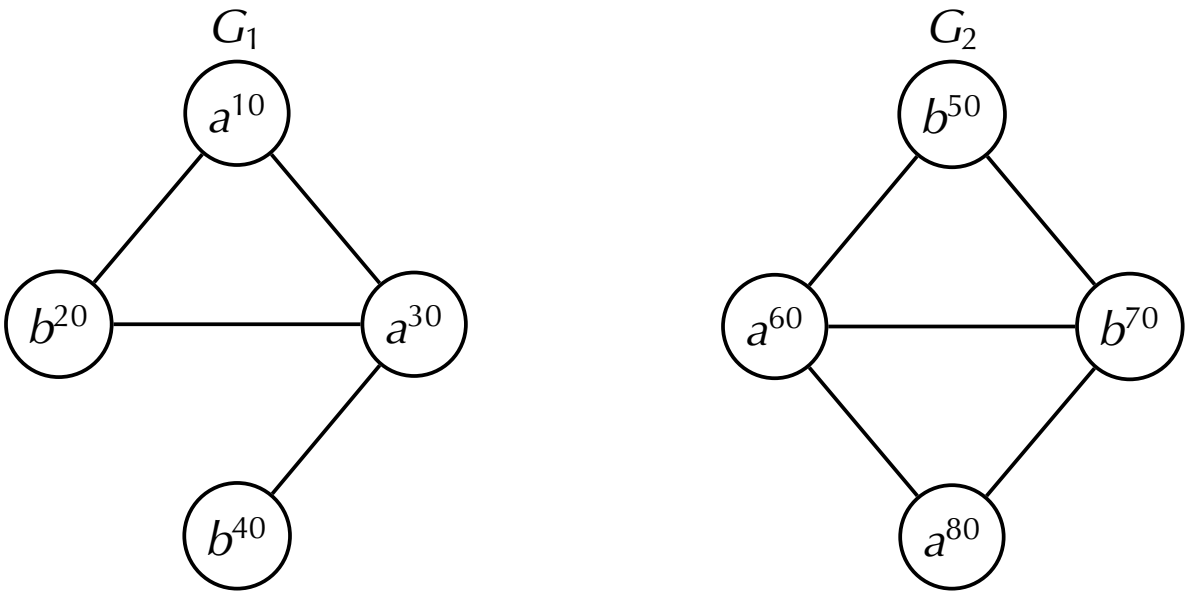
The function RIGHTMOSTPATH-EXTENSIONS returns the set of edge extensions along with their support values, $\mathcal{E}$. Each extended edge t in $\mathcal{E}$ leads to a new candidate DFS code $C' = C \cup \{t\}$, with support $sup(C') = sup(t)$.

For each new candidate code, gSpan checks whether it is frequent and canonical, and if so gSpan recursively extends C' The algorithm stops when there are no more frequent and canonical extensions possible.

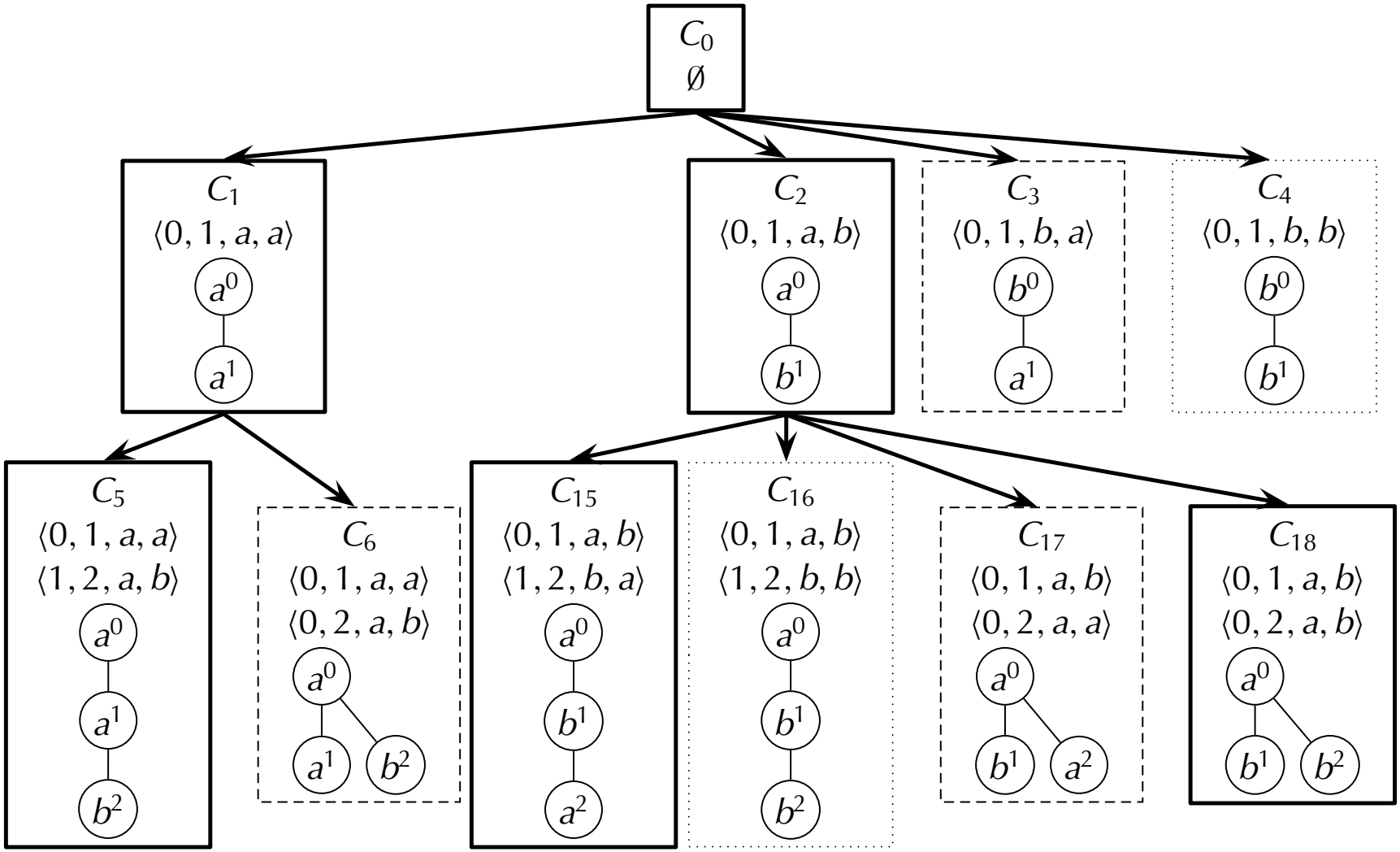
Algorithm gSpan

```
// Initial Call: C ← ∅
gSpan (C, D, minsup):
1 E ← RightMostPath-Extensions(C, D) // extensions and
      supports
2 foreach (t, sup(t)) ∈ E do
3     C ← C ∪ t // extend the code with extended edge tuple t
4     sup(C) ← sup(t) // record the support of new extension
      // recursively call gSpan if code is frequent and
         canonical
5     if sup(C) ≥ minsup and IsCanonical (C) then
6         gSpan (C, D, minsup)
```

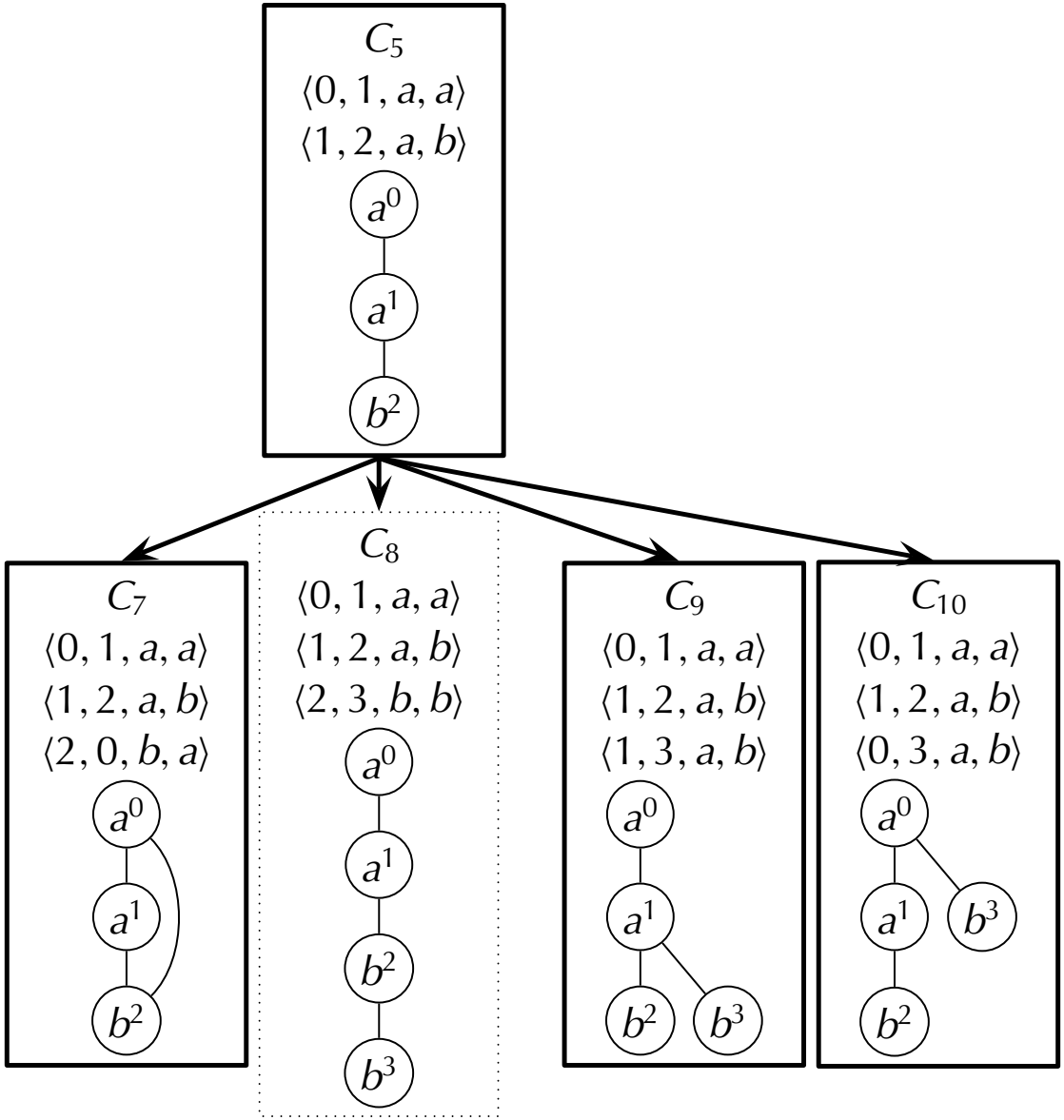
Example Graph Database: gSpan



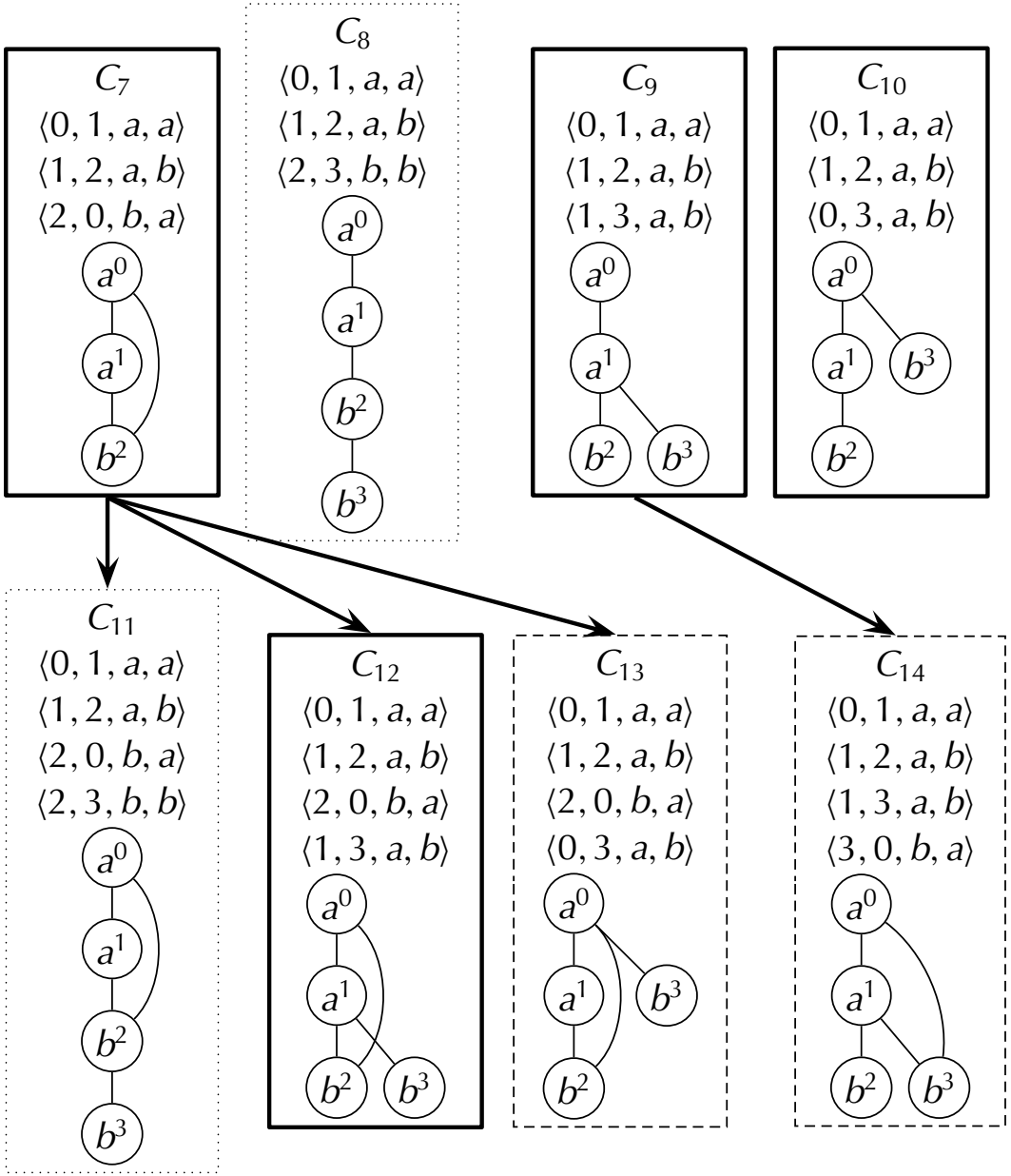
Frequent Graph Mining: gSpan



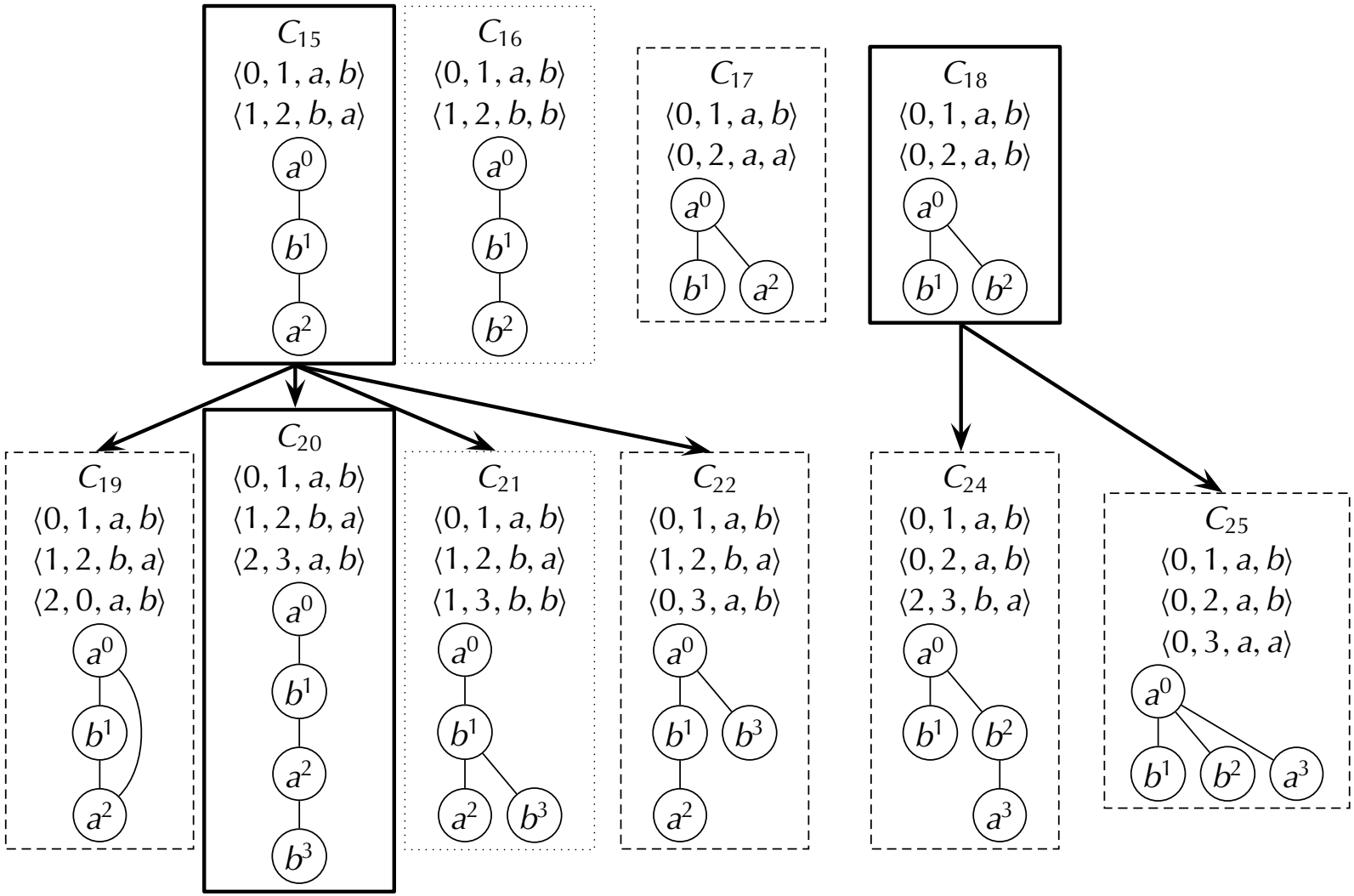
Frequent Graph Mining: gSpan



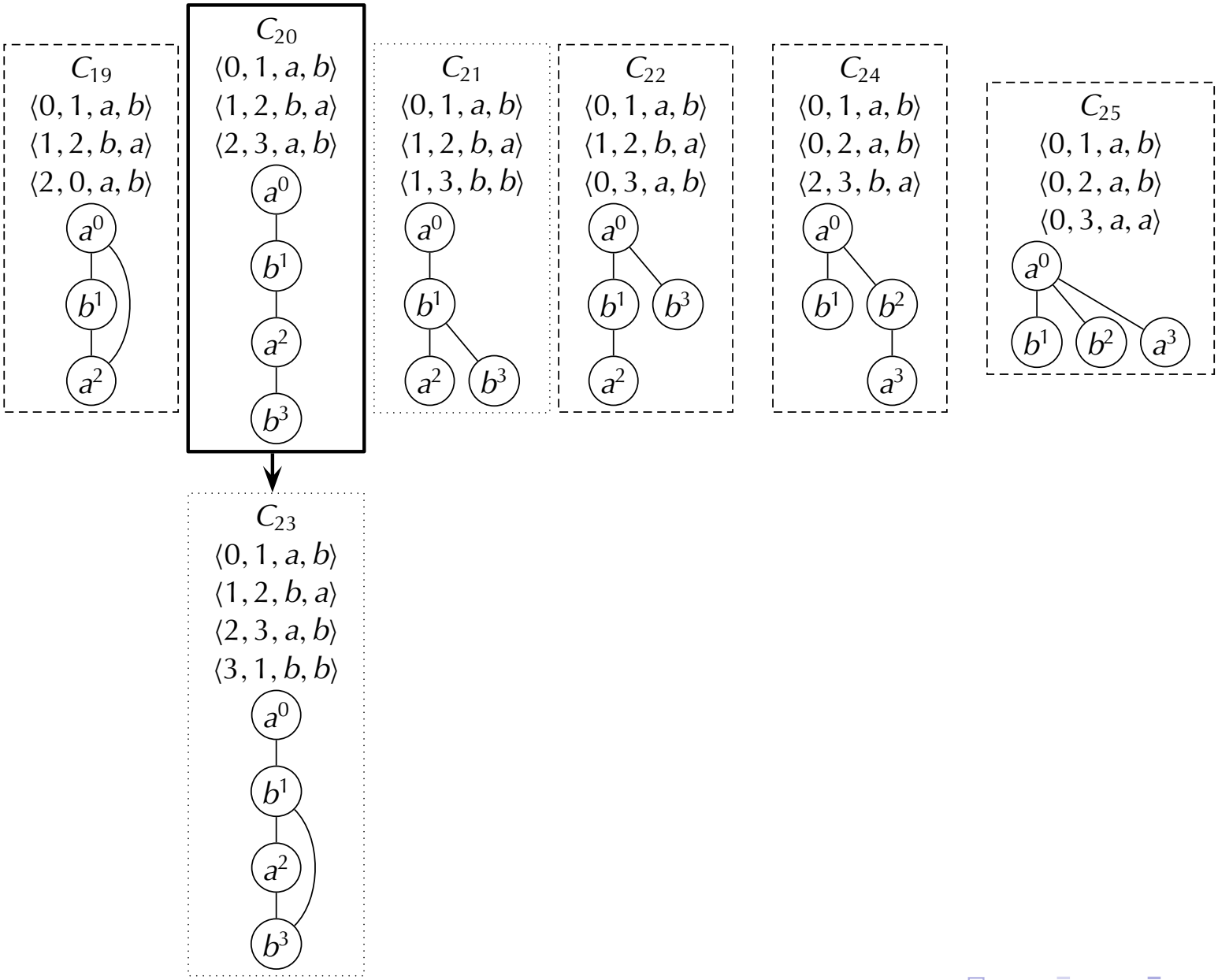
Frequent Graph Mining: gSpan



Frequent Graph Mining: gSpan



Frequent Graph Mining: gSpan



Extension and Support Computation

The support computation task is to find the number of graphs in the database $\mathbf{D}$ that contain a candidate subgraph, which is very expensive because it involves subgraph isomorphism checks. gSpan combines the tasks of enumerating candidate extensions and support computation.

Assume that $\mathbf{D} = \{G_1, G_2, \ldots, G_n\}$ comprises n graphs. Let $C = \{t_1, t_2, \ldots, t_k\}$ denote a frequent canonical DFS code comprising k edges, and let $G(C)$ denote the graph corresponding to code C. The task is to compute the set of possible rightmost path extensions from C, along with their support values.

Given code C, gSpan first records the nodes on the rightmost path (R), and the rightmost child (u_r). Next, gSpan considers each graph $G_i \in \mathbf{D}$. If $C = \emptyset$, then each distinct label tuple of the form $\langle L(x), L(y), L(x, y)\rangle$ for adjacent nodes x and y in G_i contributes a forward extension $\langle 0, 1, L(x), L(y), L(x, y)\rangle$ On the other hand, if C is not empty, then gSpan enumerates all possible subgraph isomorphisms Φ_i between the code C and graph G_i. Given subgraph isomorphism $\phi \in \Phi_i$, gSpan finds all possible forward and backward edge extensions, and stores them in the extension set $\mathcal{E}$.

Forward and Backward Extensions

Backward extensions are allowed only from the rightmost child u_r in C to some other node on the rightmost path R.

The method considers each neighbor x of $\phi(u_r)$ in G_i and checks whether it is a mapping for some vertex $v = \phi^{-1}(x)$ along the rightmost path R in C. If the edge (u_r, v) does not already exist in C, it is a new extension, and the extended tuple $b = \langle u_r, v, L(u_r), L(v), L(u_r, v) \rangle$ is added to the set of extensions $\mathcal{E}$, along with the graph id i that contributed to that extension.

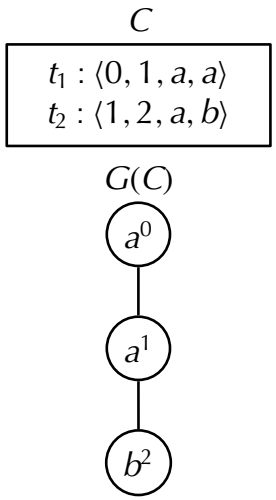
Forward extensions are allowed only from nodes on the rightmost path R to new nodes. For each node u in R, the algorithm finds a neighbor x in G_i that is not in a mapping from some node in C. For each such node x, the forward extension $f = \langle u, u_r + 1, L(\phi(u)), L(x), L(\phi(u), x) \rangle$ is added to $\mathcal{E}$, along with the graph id i. Because a forward extension adds a new vertex to the graph $G(C)$, the id of the new node in C must be $u_r + 1$, that is, one more than the highest numbered node in C, which by definition is the rightmost child u_r.

Algorithm RIGHTMOSTPATH-EXTENSIONS

RIGHTMOSTPATH-EXTENSIONS $(C, \mathbf{D})$:

$R \leftarrow$ nodes on the rightmost path in C
$u_r \leftarrow$ rightmost child in C `// dfs number`
$\mathcal{E} \leftarrow \emptyset$ `// set of extensions from` C
foreach $G_i \in \mathbf{D}$, $i = 1, \ldots, n$ **do**
  if $C = \emptyset$ **then**
    foreach *distinct* $\langle L(x), L(y), L(x,y) \rangle \in G_i$ **do**
      $f = \langle 0, 1, L(x), L(y), L(x,y) \rangle$
      Add tuple f to $\mathcal{E}$ along with graph id i
  else
    $\Phi_i =$ SUBGRAPHISOMORPHISMS (C, G_i)
    foreach *isomorphism* $\phi \in \Phi_i$ **do**
      foreach $x \in N_{G_i}(\phi(u_r))$ *such that* $\exists v \leftarrow \phi^{-1}(x)$ **do**
        if $v \in R$ *and* $(u_r, v) \notin G(C)$ **then**
          $b = \langle u_r, v, L(u_r), L(v), L(u_r, v) \rangle$
          Add tuple b to $\mathcal{E}$ along with graph id i
      foreach $u \in R$ **do**
        foreach $x \in N_{G_i}(\phi(u))$ *and* $\nexists \phi^{-1}(x)$ **do**
          $f = \langle u, u_r + 1, L(\phi(u)), L(x), L(\phi(u), x) \rangle$
          Add tuple f to $\mathcal{E}$ along with graph id i
foreach *distinct extension* $s \in \mathcal{E}$ **do**
  $sup(s) =$ number of distinct graph ids that support tuple s
return *set of pairs* $\langle s, sup(s) \rangle$ *for extensions* $s \in \mathcal{E}$, *in tuple sorted order*

Righmost Path Extensions



(a) Code C and graph $G(C)$

Φ	ϕ	0	1	2
Φ_1	ϕ_1	10	30	20
	ϕ_2	10	30	40
	ϕ_3	30	10	20
Φ_2	ϕ_4	60	80	70
	ϕ_5	80	60	50
	ϕ_6	80	60	70

(b) Subgraph isomorphisms

Id	ϕ	Extensions
G_1	ϕ_1	$\{\langle 2, 0, b, a \rangle, \langle 1, 3, a, b \rangle\}$
	ϕ_2	$\{\langle 1, 3, a, b \rangle, \langle 0, 3, a, b \rangle\}$
	ϕ_3	$\{\langle 2, 0, b, a \rangle, \langle 0, 3, a, b \rangle\}$
G_2	ϕ_4	$\{\langle 2, 0, b, a \rangle, \langle 2, 3, b, b \rangle, \langle 0, 3, a, b \rangle\}$
	ϕ_5	$\{\langle 2, 3, b, b \rangle, \langle 1, 3, a, b \rangle\}$
	ϕ_6	$\{\langle 2, 0, b, a \rangle, \langle 2, 3, b, b \rangle, \langle 1, 3, a, b \rangle\}$

(c) Edge extensions

Extension	Support
$\langle 2, 0, b, a \rangle$	2
$\langle 2, 3, b, b \rangle$	1
$\langle 1, 3, a, b \rangle$	2
$\langle 0, 3, a, b \rangle$	2

(d) Extensions (sorted) and supports

Algorithm SUBGRAPHISOMORPHISMS

SUBGRAPHISOMORPHISMS ($C = \{t_1, t_2, \ldots, t_k\}$, G):

$\Phi \leftarrow \{\phi(0) \rightarrow x \mid x \in G \text{ and } L(x) = L(0)\}$
foreach $t_i \in C$, $i = 1, \ldots, k$ **do**
    $\langle u, v, L(u), L(v), L(u,v)\rangle \leftarrow t_i$ // expand extended edge t_i
    $\Phi' \leftarrow \emptyset$ // partial isomorphisms including t_i
    foreach *partial isomorphism* $\phi \in \Phi$ **do**
        if $v > u$ **then**
            // forward edge
            foreach $x \in N_G(\phi(u))$ **do**
                if $\nexists \phi^{-1}(x)$ *and* $L(x) = L(v)$ *and* $L(\phi(u), x) = L(u, v)$ **then**
                    $\phi' \leftarrow \phi \cup \{\phi(v) \rightarrow x\}$
                    Add ϕ' to Φ'
        else
            // backward edge
            if $\phi(v) \in N_{G_j}(\phi(u))$ **then** Add ϕ to Φ' // valid isomorphism
    $\Phi \leftarrow \Phi'$ // update partial isomorphisms
return Φ

Subgraph Isomorphisms

To enumerate all the possible isomorphisms from C to each graph $G_i \in \mathbf{D}$ the function SUBGRAPHISOMORPHISMS, accepts a code C and a graph G, and returns the set of all isomorphisms between C and G.

The set of isomorphisms Φ is initialized by mapping vertex 0 in C to each vertex x in G that shares the same label as 0, that is, if $L(x) = L(0)$.

The method considers each tuple t_i in C and extends the current set of partial isomorphisms. Let $t_i = \langle u, v, L(u), L(v), L(u, v) \rangle$. We have to check if each isomorphism $\phi \in \Phi$ can be extended in G using the information from t_i

If t_i is a forward edge, then we seek a neighbor x of $\phi(u)$ in G such that x has not already been mapped to some vertex in C, that is, $\phi^{-1}(x)$ should not exist, and the node and edge labels should match, that is, $L(x) = L(v)$, and $L(\phi(u), x) = L(u, v)$. If so, ϕ can be extended with the mapping $\phi(v) \rightarrow x$. The new extended isomorphism, denoted ϕ', is added to the initially empty set of isomorphisms Φ'.

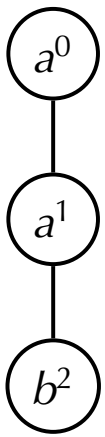
If t_i is a backward edge, we have to check if $\phi(v)$ is a neighbor of $\phi(u)$ in G. If so, we add the current isomorphism ϕ to Φ'.

Subgraph Isomorphisms

C

$t_1 : \langle 0, 1, a, a \rangle$
$t_2 : \langle 1, 2, a, b \rangle$

$G(C)$



Initial Φ

id	ϕ	0
G_1	ϕ_1	10
	ϕ_2	30
G_2	ϕ_3	60
	ϕ_4	80

Add t_1

id	ϕ	0, 1
G_1	ϕ_1	10, 30
	ϕ_2	30, 10
G_2	ϕ_3	60, 80
	ϕ_4	80, 60

Add t_2

id	ϕ	0, 1, 2
G_1	ϕ'_1	10, 30, 20
	ϕ''_1	10, 30, 40
	ϕ_2	30, 10, 20
G_2	ϕ_3	60, 80, 70
	ϕ'_4	80, 60, 50
	ϕ''_4	80, 60, 70

Canonicality Checking

Given a DFS code $C = \{t_1, t_2, \ldots, t_k\}$ comprising k extended edge tuples and the corresponding graph $G(C)$, the task is to check whether the code C is canonical.

This can be accomplished by trying to reconstruct the canonical code C^* for $G(C)$ in an iterative manner starting from the empty code and selecting the least rightmost path extension at each step, where the least edge extension is based on the extended tuple comparison operator.

If at any step the current (partial) canonical DFS code C^* is smaller than C, then we know that C cannot be canonical and can thus be pruned. On the other hand, if no smaller code is found after k extensions then C must be canonical.

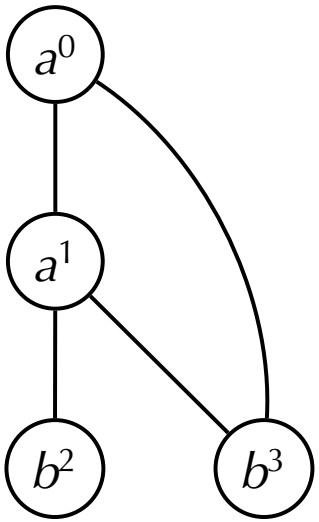
Algorithm IsCanonical: Canonicality Checking

IsCanonical (C):

1 $\mathbf{D}_C \leftarrow \{G(C)\}$ // graph corresponding to code C
2 $C^* \leftarrow \emptyset$ // initialize canonical DFScode
3 **for** $i = 1 \cdots k$ **do**
4 $\quad \mathcal{E} = \text{RightMostPath-Extensions}(C^*, \mathbf{D}_C)$ // extensions of C^*
5 $\quad (s_i, sup(s_i)) \leftarrow \min\{\mathcal{E}\}$ // least rightmost edge extension of C^*
6 $\quad$ **if** $s_i < t_i$ **then**
7 $\quad\quad$ **return** *false* // C^* is smaller, thus C is not canonical
8 $\quad C^* \leftarrow C^* \cup s_i$
9 **return** *true* // no smaller code exists; C is canonical

Canonicality Checking

G



C

$t_1 = \langle 0, 1, a, a \rangle$
$t_2 = \langle 1, 2, a, b \rangle$
$t_3 = \langle 1, 3, a, b \rangle$
$t_4 = \langle 3, 0, b, a \rangle$

Step 1

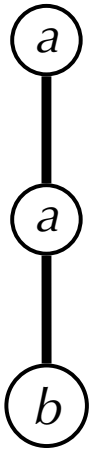
G^*

C^*

$s_1 = \langle 0, 1, a, a \rangle$

Step 2

G^*

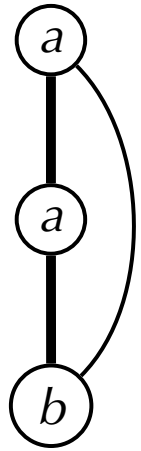


C^*

$s_1 = \langle 0, 1, a, a \rangle$
$s_2 = \langle 1, 2, a, b \rangle$

Step 3

G^*



C^*

$s_1 = \langle 0, 1, a, a \rangle$
$s_2 = \langle 1, 2, a, b \rangle$
$s_3 = \langle 2, 0, b, a \rangle$

- Frequent Subgraph Mining
- **Spectral Clustering**

Graphs and Matrices: Adjacency Matrix

Given a dataset $\mathbf{D} = \{\mathbf{x}_i\}_{i=1}^n$ consisting of n points in $\mathbb{R}^d$, let $\mathbf{A}$ denote the $n \times n$ symmetric *similarity matrix* between the points, given as

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \cdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}$$

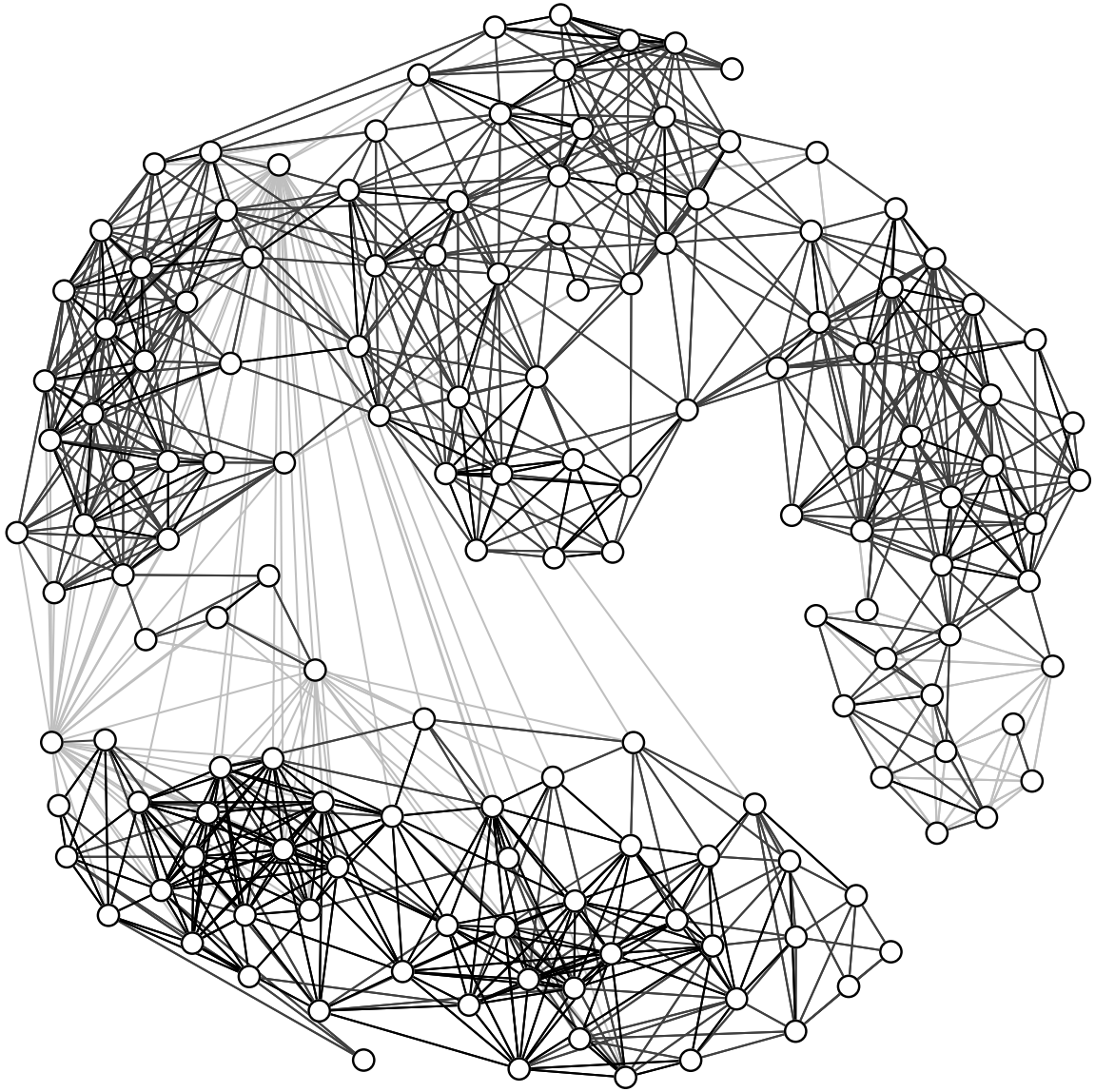
where $\mathbf{A}(i, j) = a_{ij}$ denotes the similarity or affinity between points $\mathbf{x}_i$ and $\mathbf{x}_j$.

We require the similarity to be symmetric and non-negative, that is, $a_{ij} = a_{ji}$ and $a_{ij} \geq 0$, respectively.

The matrix $\mathbf{A}$ is the *weighted adjacency matrix* for the data graph. If all affinities are 0 or 1, then $\mathbf{A}$ represents the regular adjacency relationship between the vertices.

Iris Similarity Graph: Mutual Nearest Neighbors

$|V| = n = 150, |E| = m = 1730$



Graphs and Matrices: Degree Matrix

For a vertex $\mathbf{x}_i$, let d_i denote the *degree* of the vertex, defined as

$$d_i = \sum_{j=1}^{n} a_{ij}$$

We define the *degree matrix* $\mathbf{\Delta}$ of graph G as the $n \times n$ diagonal matrix:

$$\mathbf{\Delta} = \begin{pmatrix} d_1 & 0 & \cdots & 0 \\ 0 & d_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & d_n \end{pmatrix} = \begin{pmatrix} \sum_{j=1}^{n} a_{1j} & 0 & \cdots & 0 \\ 0 & \sum_{j=1}^{n} a_{2j} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sum_{j=1}^{n} a_{nj} \end{pmatrix}$$

$\mathbf{\Delta}$ can be compactly written as $\mathbf{\Delta}(i, i) = d_i$ for all $1 \leq i \leq n$.

Graphs and Matrices: Normalized Adjacency Matrix

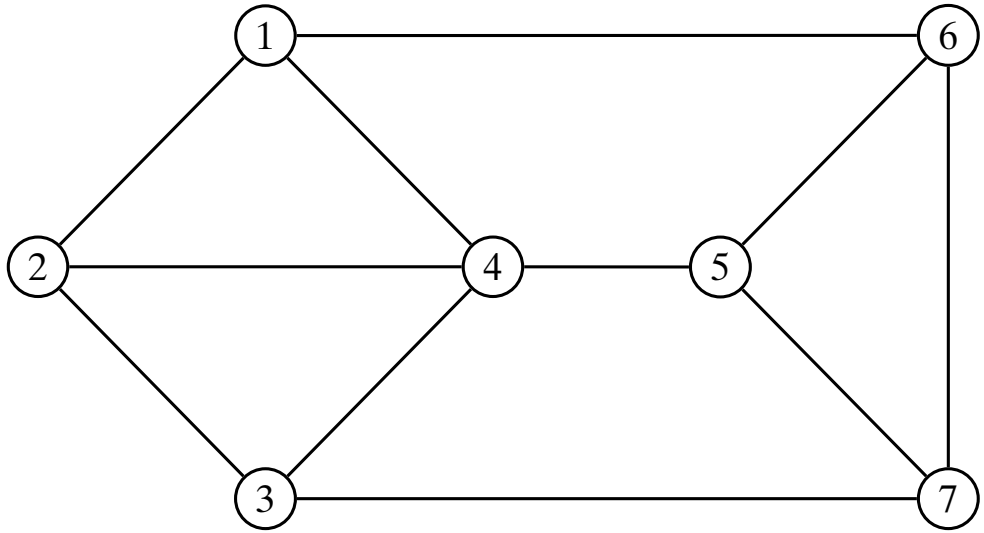
The normalized adjacency matrix is obtained by dividing each row of the adjacency matrix by the degree of the corresponding node. Given the weighted adjacency matrix $\mathbf{A}$ for a graph G, its normalized adjacency matrix is defined as

$$\mathbf{M} = \boldsymbol{\Delta}^{-1}\mathbf{A} = \begin{pmatrix} \frac{a_{11}}{d_1} & \frac{a_{12}}{d_1} & \cdots & \frac{a_{1n}}{d_1} \\ \frac{a_{21}}{d_2} & \frac{a_{22}}{d_2} & \cdots & \frac{a_{2n}}{d_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{a_{n1}}{d_n} & \frac{a_{n2}}{d_n} & \cdots & \frac{a_{nn}}{d_n} \end{pmatrix}$$

Because $\mathbf{A}$ is assumed to have non-negative elements, this implies that each element of $\mathbf{M}$, namely m_{ij} is also non-negative, as $m_{ij} = \frac{a_{ij}}{d_i} \geq 0$.

Each row in $\mathbf{M}$ sums to 1, which implies that 1 is an eigenvalue of $\mathbf{M}$. In fact, $\lambda_1 = 1$ is the largest eigenvalue of $\mathbf{M}$, and the other eigenvalues satisfy the property that $|\lambda_i| \leq 1$. Because $\mathbf{M}$ is not symmetric, its eigenvectors are not necessarily orthogonal.

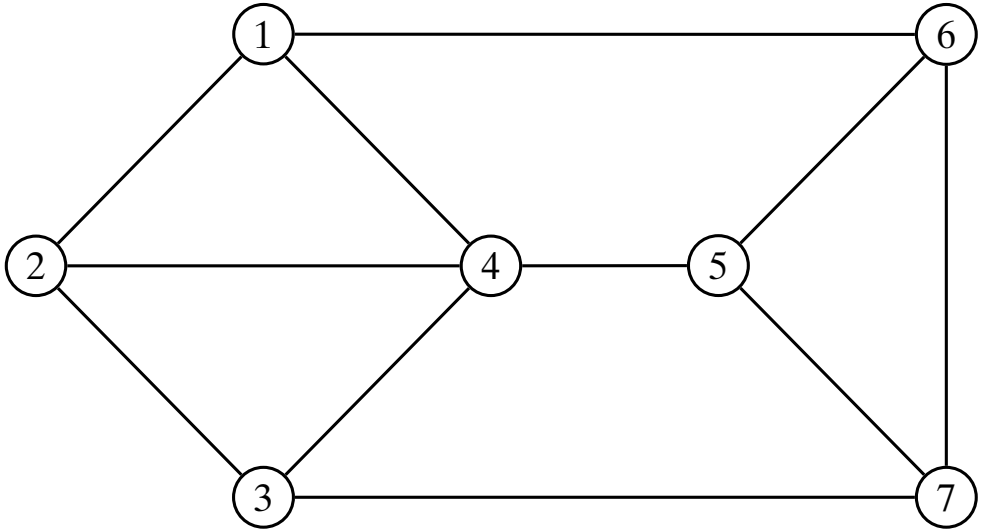
Example Graph: Adjacency and Degree Matrices



Its adjacency and degree matrices are given as

$$\mathbf{A} = \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \end{pmatrix} \qquad \mathbf{\Delta} = \begin{pmatrix} 3 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 3 \end{pmatrix}$$

Example Graph: Normalized Adjacency Matrix



The normalized adjacency matrix is as follows:

$$\mathbf{M} = \mathbf{\Delta}^{-1}\mathbf{A} = \begin{pmatrix} 0 & 0.33 & 0 & 0.33 & 0 & 0.33 & 0 \\ 0.33 & 0 & 0.33 & 0.33 & 0 & 0 & 0 \\ 0 & 0.33 & 0 & 0.33 & 0 & 0 & 0.33 \\ 0.25 & 0.25 & 0.25 & 0 & 0.25 & 0 & 0 \\ 0 & 0 & 0 & 0.33 & 0 & 0.33 & 0.33 \\ 0.33 & 0 & 0 & 0 & 0.33 & 0 & 0.33 \\ 0 & 0 & 0.33 & 0 & 0.33 & 0.33 & 0 \end{pmatrix}$$

The eigenvalues of $\mathbf{M}$ are: $\lambda_1 = 1, \lambda_2 = 0.483, \lambda_3 = 0.206, \lambda_4 = -0.045, \lambda_5 = -0.405, \lambda_6 = -0.539, \lambda_7 = -0.7$

Graph Laplacian Matrix

The *Laplacian matrix* of a graph is defined as

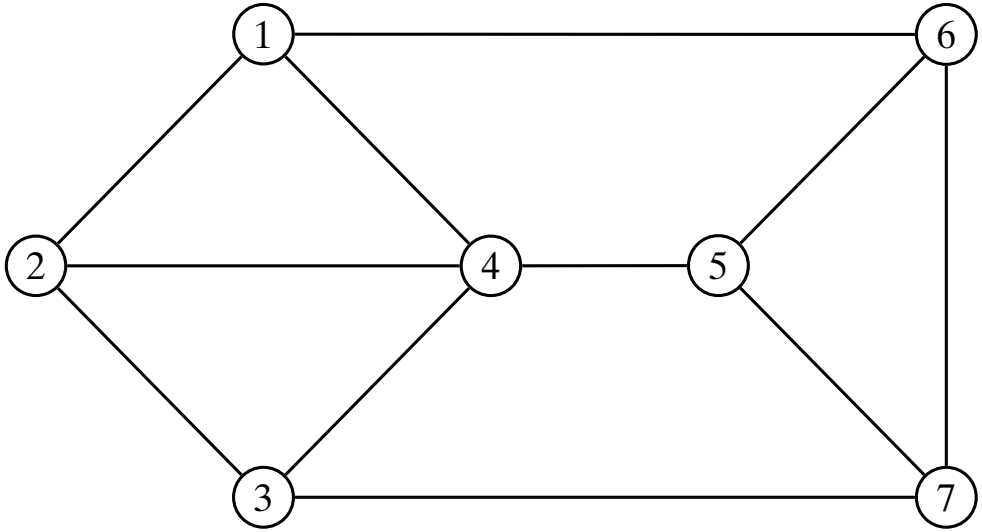
$$\mathbf{L} = \mathbf{\Delta} - \mathbf{A}$$

$$= \begin{pmatrix} \sum_{j=1}^{n} a_{1j} & 0 & \cdots & 0 \\ 0 & \sum_{j=1}^{n} a_{2j} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sum_{j=1}^{n} a_{nj} \end{pmatrix} - \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \cdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}$$

$$= \begin{pmatrix} \sum_{j \neq 1} a_{1j} & -a_{12} & \cdots & -a_{1n} \\ -a_{21} & \sum_{j \neq 2} a_{2j} & \cdots & -a_{2n} \\ \vdots & \vdots & \cdots & \vdots \\ -a_{n1} & -a_{n2} & \cdots & \sum_{j \neq n} a_{nj} \end{pmatrix}$$

$\mathbf{L}$ is a symmetric, positive semidefinite matrix. This means that $\mathbf{L}$ has n real, non-negative eigenvalues, which can be arranged in decreasing order as follows: $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n \geq 0$. Because $\mathbf{L}$ is symmetric, its eigenvectors are orthonormal. The rank of $\mathbf{L}$ is at most $n-1$, and the smallest eigenvalue is $\lambda_n = 0$.

Example Graph: Laplacian Matrix



The graph Laplacian is given as

$$\mathbf{L} = \mathbf{\Delta} - \mathbf{A} = \begin{pmatrix} 3 & -1 & 0 & -1 & 0 & -1 & 0 \\ -1 & 3 & -1 & -1 & 0 & 0 & 0 \\ 0 & -1 & 3 & -1 & 0 & 0 & -1 \\ -1 & -1 & -1 & 4 & -1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 3 & -1 & -1 \\ -1 & 0 & 0 & 0 & -1 & 3 & -1 \\ 0 & 0 & -1 & 0 & -1 & -1 & 3 \end{pmatrix}$$

The eigenvalues of $\mathbf{L}$ are as follows: $\lambda_1 = 5.618, \lambda_2 = 4.618, \lambda_3 = 4.414, \lambda_4 = 3.382, \lambda_5 = 2.382, \lambda_6 = 1.586, \lambda_7 = 0$

Normalized Laplacian Matrices

The *normalized symmetric Laplacian matrix* of a graph is defined as

$$\mathbf{L}^s = \mathbf{\Delta}^{-1/2}\mathbf{L}\mathbf{\Delta}^{-1/2} = \begin{pmatrix} \frac{\sum_{j\neq 1} a_{1j}}{\sqrt{d_1 d_1}} & -\frac{a_{12}}{\sqrt{d_1 d_2}} & \cdots & -\frac{a_{1n}}{\sqrt{d_1 d_n}} \\ -\frac{a_{21}}{\sqrt{d_2 d_1}} & \frac{\sum_{j\neq 2} a_{2j}}{\sqrt{d_2 d_2}} & \cdots & -\frac{a_{2n}}{\sqrt{d_2 d_n}} \\ \vdots & \vdots & \ddots & \vdots \\ -\frac{a_{n1}}{\sqrt{d_n d_1}} & -\frac{a_{n2}}{\sqrt{d_n d_2}} & \cdots & \frac{\sum_{j\neq n} a_{nj}}{\sqrt{d_n d_n}} \end{pmatrix}$$

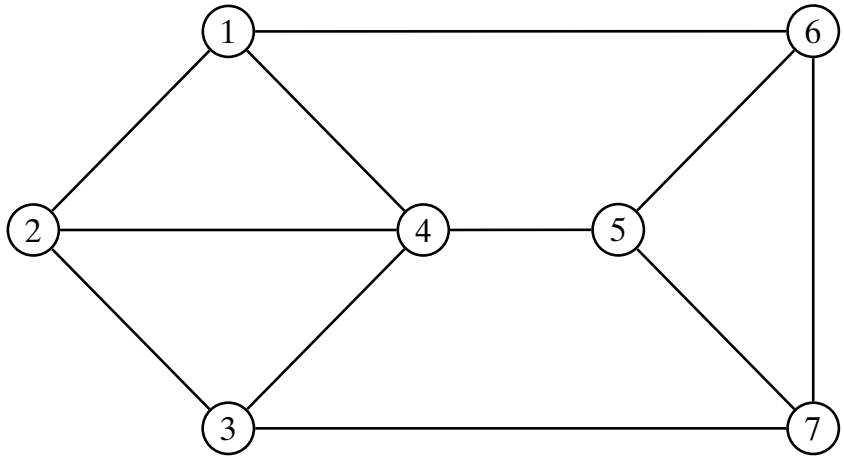
$\mathbf{L}^s$ is a symmetric, positive semidefinite matrix, with rank at most $n-1$. The smallest eigenvalue $\lambda_n = 0$.

The *normalized asymmetric Laplacian* matrix is defined as

$$\mathbf{L}^a = \mathbf{\Delta}^{-1}\mathbf{L} = \begin{pmatrix} \frac{\sum_{j\neq 1} a_{1j}}{d_1} & -\frac{a_{12}}{d_1} & \cdots & -\frac{a_{1n}}{d_1} \\ -\frac{a_{21}}{d_2} & \frac{\sum_{j\neq 2} a_{2j}}{d_2} & \cdots & -\frac{a_{2n}}{d_2} \\ \vdots & \vdots & \ddots & \vdots \\ -\frac{a_{n1}}{d_n} & -\frac{a_{n2}}{d_n} & \cdots & \frac{\sum_{j\neq n} a_{nj}}{d_n} \end{pmatrix}$$

$\mathbf{L}^a$ is also a positive semi-definite matrix with n real eigenvalues $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n = 0$.

Example Graph: Normalized Symmetric Laplacian Matrix

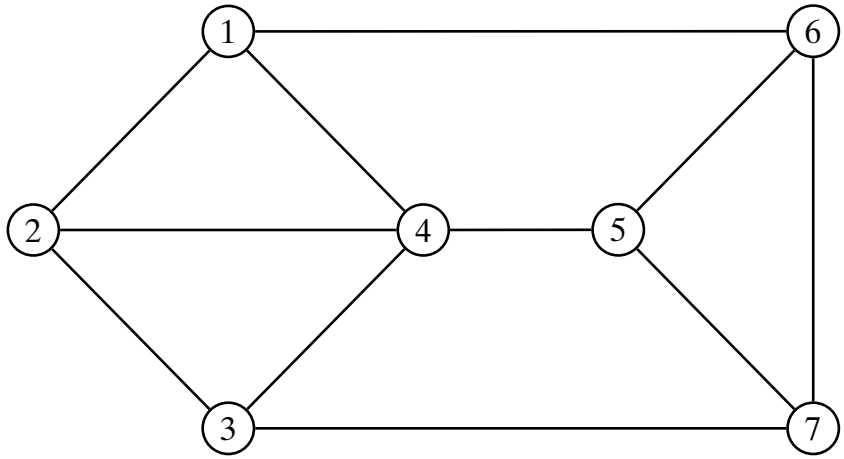


The normalized symmetric Laplacian is given as

$$\mathbf{L}^s = \begin{pmatrix} 1 & -0.33 & 0 & -0.29 & 0 & -0.33 & 0 \\ -0.33 & 1 & -0.33 & -0.29 & 0 & 0 & 0 \\ 0 & -0.33 & 1 & -0.29 & 0 & 0 & -0.33 \\ -0.29 & -0.29 & -0.29 & 1 & -0.29 & 0 & 0 \\ 0 & 0 & 0 & -0.29 & 1 & -0.33 & -0.33 \\ -0.33 & 0 & 0 & 0 & -0.33 & 1 & -0.33 \\ 0 & 0 & -0.33 & 0 & -0.33 & -0.33 & 1 \end{pmatrix}$$

The eigenvalues of $\mathbf{L}^s$ are as follows: $\lambda_1 = 1.7, \lambda_2 = 1.539, \lambda_3 = 1.405, \lambda_4 = 1.045, \lambda_5 = 0.794, \lambda_6 = 0.517, \lambda_7 = 0$

Example Graph: Normalized Asymmetric Laplacian Matrix



The normalized asymmetric Laplacian matrix is given as

$$\mathbf{L}^a = \mathbf{\Delta}^{-1}\mathbf{L} = \begin{pmatrix} 1 & -0.33 & 0 & -0.33 & 0 & -0.33 & 0 \\ -0.33 & 1 & -0.33 & -0.33 & 0 & 0 & 0 \\ 0 & -0.33 & 1 & -0.33 & 0 & 0 & -0.33 \\ -0.25 & -0.25 & -0.25 & 1 & -0.25 & 0 & 0 \\ 0 & 0 & 0 & -0.33 & 1 & -0.33 & -0.33 \\ -0.33 & 0 & 0 & 0 & -0.33 & 1 & -0.33 \\ 0 & 0 & -0.33 & 0 & -0.33 & -0.33 & 1 \end{pmatrix}$$

The eigenvalues of $\mathbf{L}^a$ are identical to those for $\mathbf{L}^s$, namely $\lambda_1 = 1.7, \lambda_2 = 1.539, \lambda_3 = 1.405, \lambda_4 = 1.045, \lambda_5 = 0.794, \lambda_6 = 0.517, \lambda_7 = 0$

Clustering as Graph Cuts

A *k-way cut* in a graph is a partitioning or clustering of the vertex set, given as $\mathcal{C} = \{C_1, \ldots, C_k\}$. We require $\mathcal{C}$ to optimize some objective function that captures the intuition that nodes within a cluster should have high similarity, and nodes from different clusters should have low similarity.

Given a weighted graph G defined by its similarity matrix $\mathbf{A}$, let $S, T \subseteq V$ be any two subsets of the vertices. We denote by $W(S, T)$ the sum of the weights on all edges with one vertex in S and the other in T, given as

$$W(S, T) = \sum_{v_i \in S} \sum_{v_j \in T} a_{ij}$$

Given $S \subseteq V$, we denote by $\overline{S}$ the complementary set of vertices, that is, $\overline{S} = V - S$. A *(vertex) cut* in a graph is defined as a partitioning of V into $S \subset V$ and $\overline{S}$. The *weight of the cut* or *cut weight* is defined as the sum of all the weights on edges between vertices in S and $\overline{S}$, given as $W(S, \overline{S})$.

Cuts and Matrix Operations

Given a clustering $\mathcal{C} = \{C_1, \ldots, C_k\}$ comprising k clusters. Let $\mathbf{c}_i \in \{0,1\}^n$ be the *cluster indicator vector* that records the cluster membership for cluster C_i, defined as

$$c_{ij} = \begin{cases} 1 & \text{if } v_j \in C_i \\ 0 & \text{if } v_j \notin C_i \end{cases}$$

The cluster size can be written as

$$|C_i| = \mathbf{c}_i^T \mathbf{c}_i = \|\mathbf{c}_i\|^2$$

The *volume* of a cluster C_i is defined as the sum of all the weights on edges with one end in cluster C_i:

$$vol(C_i) = W(C_i, V) = \sum_{v_r \in C_i} d_r = \sum_{v_r \in C_i} c_{ir} d_r c_{ir} = \sum_{r=1}^{n} \sum_{s=1}^{n} c_{ir} \Delta_{rs} c_{is} = \mathbf{c}_i^T \mathbf{\Delta} \mathbf{c}_i$$

The sum of weights of all internal edges is:

$$W(C_i, C_i) = \sum_{v_r \in C_i} \sum_{v_s \in C_i} a_{rs} = \sum_{r=1}^{n} \sum_{s=1}^{n} c_{ir} a_{rs} c_{is} = \mathbf{c}_i^T \mathbf{A} \mathbf{c}_i$$

We can get the sum of weights for all the external edges as follows:

$$W(C_i, \overline{C_i}) = \sum_{v_r \in C_i} \sum_{v_s \in V - C_i} a_{rs} = W(C_i, V) - W(C_i, C_i) = \mathbf{c}_i(\mathbf{\Delta} - \mathbf{A})\mathbf{c}_i = \mathbf{c}_i^T \mathbf{L} \mathbf{c}_i$$

Clustering Objective Functions: Ratio Cut

The clustering objective function can be formulated as an optimization problem over the k-way cut $\mathcal{C} = \{C_1, \ldots, C_k\}$.
The *ratio cut* objective is defined over a k-way cut as follows:

$$\min_{\mathcal{C}} J_{rc}(\mathcal{C}) = \sum_{i=1}^{k} \frac{W(C_i, \overline{C_i})}{|C_i|} = \sum_{i=1}^{k} \frac{\mathbf{c}_i^T \mathbf{L} \mathbf{c}_i}{\mathbf{c}_i^T \mathbf{c}_i} = \sum_{i=1}^{k} \frac{\mathbf{c}_i^T \mathbf{L} \mathbf{c}_i}{\|\mathbf{c}_i\|^2}$$

Ratio cut tries to minimize the sum of the similarities from a cluster C_i to other points not in the cluster $\overline{C_i}$, taking into account the size of each cluster.
Unfortunately, for binary cluster indicator vectors $\mathbf{c}_i$, the ratio cut objective is NP-hard. An obvious relaxation is to allow $\mathbf{c}_i$ to take on any real value. In this case, we can rewrite the objective as

$$\min_{\mathcal{C}} J_{rc}(\mathcal{C}) = \sum_{i=1}^{k} \frac{\mathbf{c}_i^T \mathbf{L} \mathbf{c}_i}{\|\mathbf{c}_i\|^2} = \sum_{i=1}^{k} \left(\frac{\mathbf{c}_i}{\|\mathbf{c}_i\|}\right)^T \mathbf{L} \left(\frac{\mathbf{c}_i}{\|\mathbf{c}_i\|}\right) = \sum_{i=1}^{k} \mathbf{u}_i^T \mathbf{L} \mathbf{u}_i$$

The optimal solution comprises the eigenvectors corresponding to the k smallest eigenvalues of $\mathbf{L}$, i.e., the eigenvectors $\mathbf{u}_n, \mathbf{u}_{n-1}, \ldots, \mathbf{u}_{n-k+1}$ represent the relaxed cluster indicator vectors.

Clustering Objective Functions: Normalized Cut

Normalized cut is similar to ratio cut, except that it divides the cut weight of each cluster by the volume of a cluster instead of its size. The objective function is given as

$$\min_{\mathcal{C}} J_{nc}(\mathcal{C}) = \sum_{i=1}^{k} \frac{W(C_i, \overline{C_i})}{vol(C_i)} = \sum_{i=1}^{k} \frac{\mathbf{c}_i^T \mathbf{L} \mathbf{c}_i}{\mathbf{c}_i^T \mathbf{\Delta} \mathbf{c}_i}$$

We can obtain an optimal solution by allowing $\mathbf{c}_i$ to be an arbitrary real vector.

The optimal solution comprise the eigenvectors corresponding to the k smallest eigenvalues of either the normalized symmetric or asymmetric Laplacian matrices, $\mathbf{L}^s$ and $\mathbf{L}^a$.

Spectral Clustering Algorithm

The spectral clustering algorithm takes a dataset $\mathbf{D}$ as input and computes the similarity matrix $\mathbf{A}$. For normalized cut we chose either $\mathbf{L}^s$ or $\mathbf{L}^a$, whereas for ratio cut we choose $\mathbf{L}$. Next, we compute the k smallest eigenvalues and corresponding eigenvectors of the chosen matrix.

The main problem is that the eigenvectors $\mathbf{u}_i$ are not binary, and thus it is not immediately clear how we can assign points to clusters.

One solution to this problem is to treat the $n \times k$ matrix of eigenvectors as a new data matrix:

$$\mathbf{U} = \begin{pmatrix} | & | & & | \\ \mathbf{u}_n & \mathbf{u}_{n-1} & \cdots & \mathbf{u}_{n-k+1} \\ | & | & & | \end{pmatrix} \rightarrow \text{ normalize rows } \rightarrow \begin{pmatrix} — & \mathbf{y}_1^T & — \\ — & \mathbf{y}_2^T & — \\ & \vdots & \\ — & \mathbf{y}_n^T & — \end{pmatrix} = \mathbf{Y}$$

We then cluster the new points in $\mathbf{Y}$ into k clusters via the K-means algorithm or any other fast clustering method to obtain binary cluster indicator vectors $\mathbf{c}_i$.

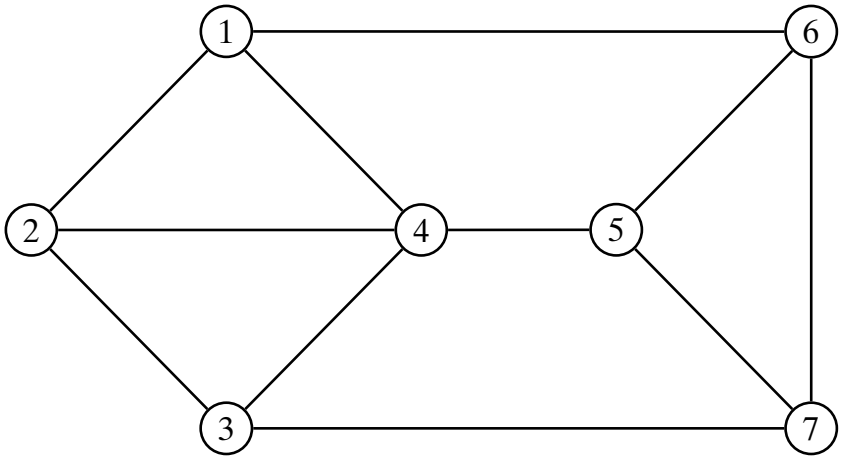
Spectral Clustering Algorithm

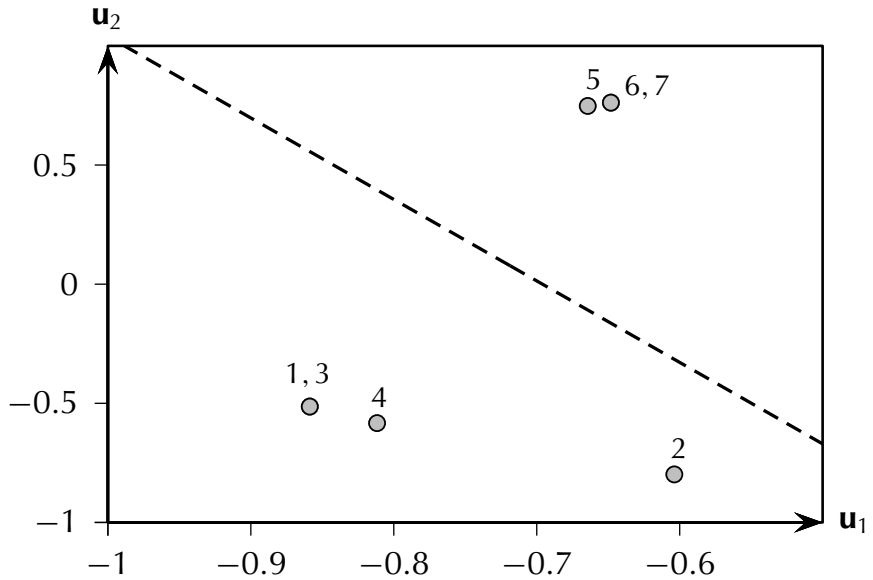
SPECTRAL CLUSTERING (D, k):

1 Compute the similarity matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$
2 **if** *ratio cut* **then** $\mathbf{B} \leftarrow \mathbf{L}$
3 **else if** *normalized cut* **then** $\mathbf{B} \leftarrow \mathbf{L}^s$ or $\mathbf{L}^a$
4 Solve $\mathbf{B}\mathbf{u}_i = \lambda_i \mathbf{u}_i$ for $i = n, \ldots, n-k+1$, where $\lambda_n \leq \lambda_{n-1} \leq \cdots \leq \lambda_{n-k+1}$
5 $\mathbf{U} \leftarrow \begin{pmatrix} \mathbf{u}_n & \mathbf{u}_{n-1} & \cdots & \mathbf{u}_{n-k+1} \end{pmatrix}$
6 $\mathbf{Y} \leftarrow$ normalize rows of $\mathbf{U}$
7 $\mathcal{C} \leftarrow \{C_1, \ldots, C_k\}$ via K-means on $\mathbf{Y}$

Spectral Clustering on Example Graph

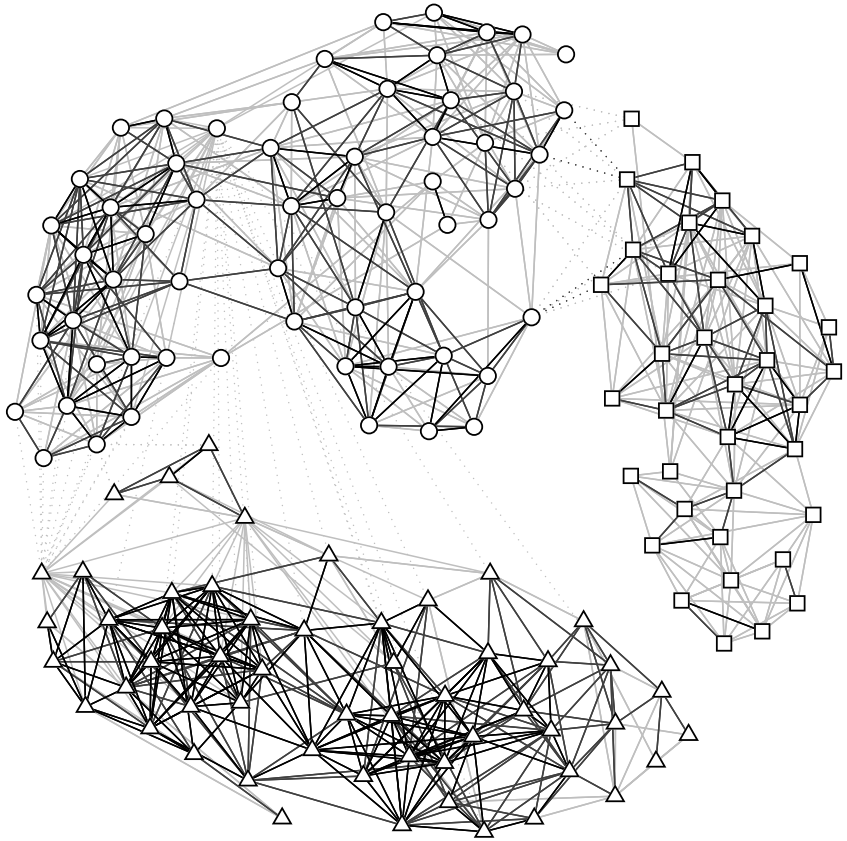
$k = 2$, normalized cut (normalized asymmetric Laplacian)





Normalized Cut on Iris Graph

$k = 3$, normalized asymmetric Laplacian



	setosa	virginica	versicolor
C_1 (triangle)	50	0	4
C_2 (square)	0	36	0
C_3 (circle)	0	14	46

Maximization Objectives: Average Cut

The *average weight* objective is defined as

$$\max_{\mathcal{C}} J_{aw}(\mathcal{C}) = \sum_{i=1}^{k} \frac{W(C_i, C_i)}{|C_i|} = \sum_{i=1}^{k} \frac{\mathbf{c}_i^T \mathbf{A} \mathbf{c}_i}{\mathbf{c}_i^T \mathbf{c}_i} = \sum_{i=1}^{k} \mathbf{u}_i^T \mathbf{A} \mathbf{u}_i$$

where $\mathbf{u}_i$ is an arbitrary real vector, which is a relaxation of the binary cluster indicator vectors $\mathbf{c}_i$.

We can maximize the objective by selecting the k largest eigenvalues of $\mathbf{A}$, and the corresponding eigenvectors.

$$\begin{aligned} \max_{\mathcal{C}} J_{aw}(\mathcal{C}) &= \mathbf{u}_1^T \mathbf{A} \mathbf{u}_1 + \cdots + \mathbf{u}_k^T \mathbf{A} \mathbf{u}_k \\ &= \lambda_1 + \cdots + \lambda_k \end{aligned}$$

where $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n$. In general, while $\mathbf{A}$ is symmetric, it may not be positive semidefinite. This means that $\mathbf{A}$ can have negative eigenvalues, and to maximize the objective we must consider only the positive eigenvalues and the corresponding eigenvectors.

Maximization Objectives: Modularity

Given $\mathbf{A}$, the weighted adjacency matrix, the modularity of a clustering is the difference between the observed and expected fraction of weights on edges within the clusters. The clustering objective is given as

$$\max_{\mathcal{C}} J_Q(\mathcal{C}) = \sum_{i=1}^{k} \left(\frac{\mathbf{c}_i^T \mathbf{A} \mathbf{c}_i}{tr(\mathbf{\Delta})} - \frac{(\mathbf{d}_i^T \mathbf{c}_i)^2}{tr(\mathbf{\Delta})^2} \right) = \sum_{i=1}^{k} \mathbf{c}_i^T \mathbf{Q} \mathbf{c}_i$$

where $\mathbf{Q}$ is the *modularity matrix*:

$$\mathbf{Q} = \frac{1}{tr(\mathbf{\Delta})} \left(\mathbf{A} - \frac{\mathbf{d} \cdot \mathbf{d}^T}{tr(\mathbf{\Delta})} \right)$$

The optimal solution comprises the eigenvectors corresponding to the k largest eigenvalues of $\mathbf{Q}$. Since $\mathbf{Q}$ is symmetric, but not positive semidefinite, we use only the positive eigenvalues.

Markov Chain Clustering

A Markov chain is a discrete-time stochastic process over a set of states, in our case the set of vertices V.

The Markov chain makes a transition from one node to another at discrete timesteps $t = 1, 2, \ldots$, with the probability of making a transition from node i to node j given as m_{ij}.

Let the random variable X_t denote the state at time t. The Markov property means that the probability distribution of X_t over the states at time t depends only on the probability distribution of X_{t-1}, that is,

$$P(X_t = i | X_0, X_1, \ldots, X_{t-1}) = P(X_t = i | X_{t-1})$$

Further, we assume that the Markov chain is *homogeneous*, that is, the transition probability

$$P(X_t = j | X_{t-1} = i) = m_{ij}$$

is independent of the time step t.

Markov Chain Clustering: Markov Matrix

The normalized adjacency matrix $\mathbf{M} = \mathbf{\Delta}^{-1}\mathbf{A}$ can be interpreted as the $n \times n$ *transition matrix* where the entry $m_{ij} = \frac{a_{ij}}{d_i}$ is the probability of transitioning or jumping from node i to node j in the graph G.

The matrix $\mathbf{M}$ is thus the transition matrix for a *Markov chain* or a Markov random walk on graph G. That is, given node i the transition matrix $\mathbf{M}$ specifies the probabilities of reaching any other node j in one time step.

In general, the transition probability matrix for t time steps is given as

$$\mathbf{M}^{t-1} \cdot \mathbf{M} = \mathbf{M}^t$$

Markov Chain Clustering: Random Walk

A random walk on G thus corresponds to taking successive powers of the transition matrix $\mathbf{M}$.

Let $\boldsymbol{\pi}_0$ specify the initial state probability vector at time $t = 0$. The state probability vector after t steps is

$$\boldsymbol{\pi}_t^T = \boldsymbol{\pi}_{t-1}^T \mathbf{M} = \boldsymbol{\pi}_{t-2}^T \mathbf{M}^2 = \cdots = \boldsymbol{\pi}_0^T \mathbf{M}^t$$

Equivalently, taking transpose on both sides, we get

$$\boldsymbol{\pi}_t = (M^t)^T \boldsymbol{\pi}_0 = (\mathbf{M}^T)^t \boldsymbol{\pi}_0$$

The state probability vector thus converges to the dominant eigenvector of $\mathbf{M}^T$.

Markov Clustering Algorithm

Consider a variation of the random walk, where the probability of transitioning from node i to j is inflated by taking each element m_{ij} to the power $r \geq 1$. Given a transition matrix $\mathbf{M}$, define the inflation operator Υ as follows:

$$\Upsilon(\mathbf{M}, r) = \left\{ \frac{(m_{ij})^r}{\sum_{a=1}^{n} (m_{ia})^r} \right\}_{i,j=1}^{n}$$

The net effect of the inflation operator is to increase the higher probability transitions and decrease the lower probability transitions.

The Markov clustering algorithm (MCL) is an iterative method that interleaves matrix expansion and inflation steps. Matrix expansion corresponds to taking successive powers of the transition matrix, leading to random walks of longer lengths. On the other hand, matrix inflation makes the higher probability transitions even more likely and reduces the lower probability transitions.

MCL takes as input the inflation parameter $r \geq 1$. Higher values lead to more, smaller clusters, whereas smaller values lead to fewer, but larger clusters.

Markov Clustering Algorithm: MCL

The final clusters are found by enumerating the weakly connected components in the directed graph induced by the converged transition matrix $\mathbf{M}_t$, where the edges are defined as:

$$E = \left\{(i,j) \mid \mathbf{M}_t(i,j) > 0\right\}$$

A directed edge (i,j) exists only if node i can transition to node j within t steps of the expansion and inflation process.

A node j is called an *attractor* if $\mathbf{M}_t(j,j) > 0$, and we say that node i is attracted to attractor j if $\mathbf{M}_t(i,j) > 0$. The MCL process yields a set of attractor nodes, $V_a \subseteq V$, such that other nodes are attracted to at least one attractor in V_a.

To extract the clusters from G_t, MCL first finds the strongly connected components $S_1, S_2, \ldots, S_q$ over the set of attractors V_a. Next, for each strongly connected set of attractors S_j, MCL finds the weakly connected components consisting of all nodes $i \in V_t - V_a$ attracted to an attractor in S_j. If a node i is attracted to multiple strongly connected components, it is added to each such cluster, resulting in possibly overlapping clusters.

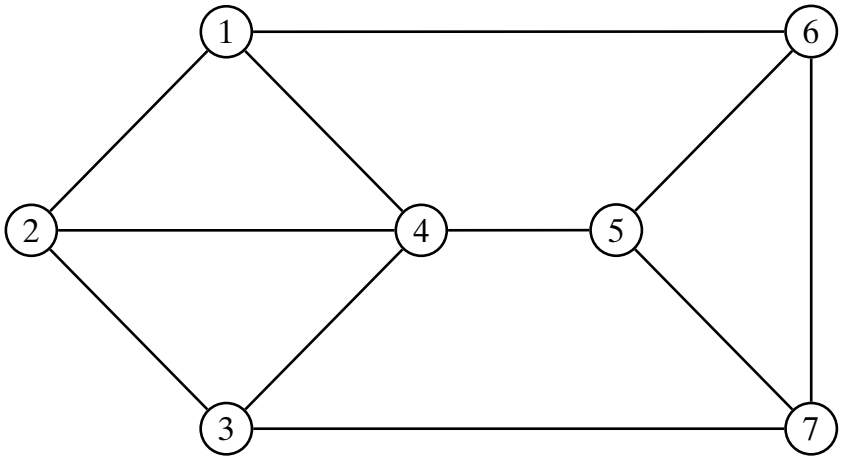
Algorithm MARKOV CLUSTERING

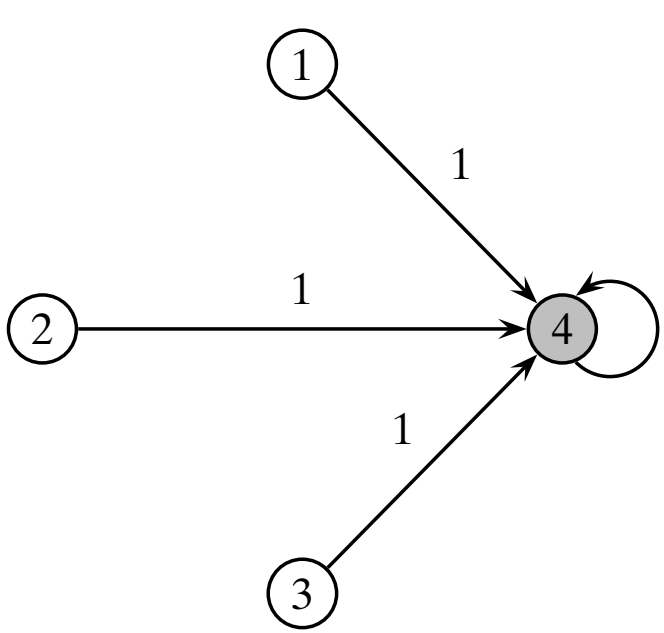
MARKOV CLUSTERING $(\mathbf{A}, r, \epsilon)$:

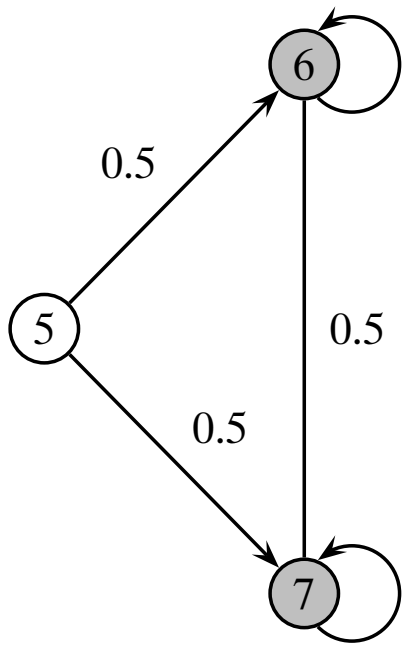
1 $t \leftarrow 0$
2 Add self-edges to $\mathbf{A}$ if they do not exist
3 $\mathbf{M}_t \leftarrow \mathbf{\Delta}^{-1}\mathbf{A}$
4 **repeat**
5 $\quad t \leftarrow t+1$
6 $\quad \mathbf{M}_t \leftarrow \mathbf{M}_{t-1} \cdot \mathbf{M}_{t-1}$
7 $\quad \mathbf{M}_t \leftarrow \Upsilon(\mathbf{M}_t, r)$
8 **until** $\|\mathbf{M}_t - \mathbf{M}_{t-1}\|_F \leq \epsilon$
9 $G_t \leftarrow$ directed graph induced by $\mathbf{M}_t$
10 $\mathcal{C} \leftarrow \{$weakly connected components in $G_t\}$

MCL Attractors and Clusters

$r = 2.5$

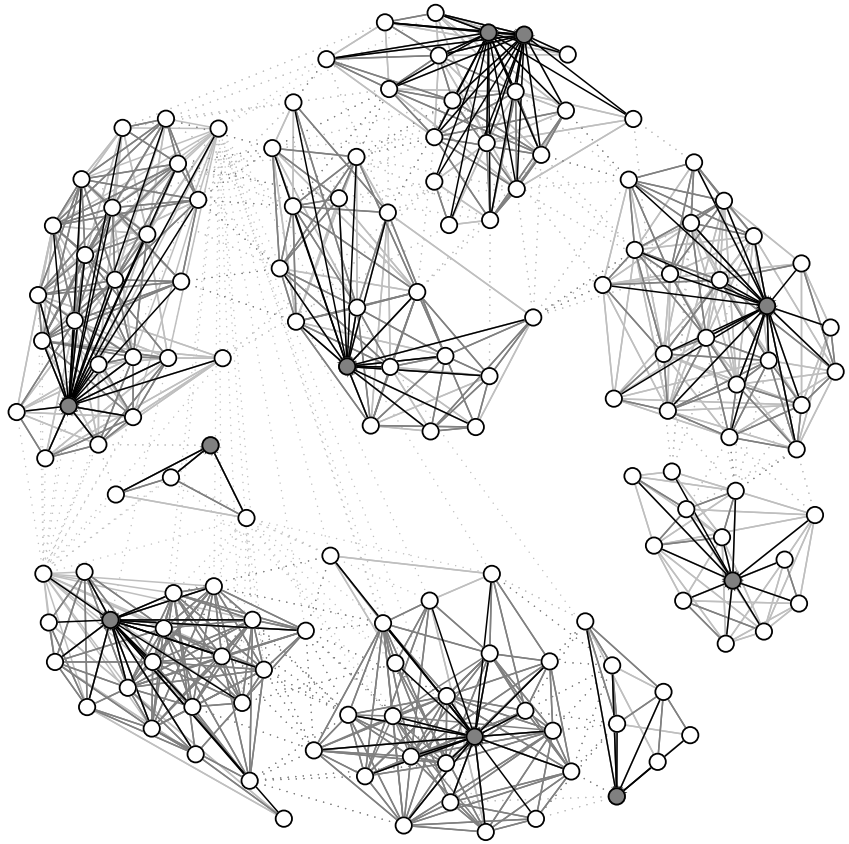






MCL on Iris Graph

(e) $r = 1.3$



(f) $r = 2$

Contingency Table: MCL Clusters versus Iris Types

$r = 1.3$

	iris-setosa	iris-virginica	iris-versicolor
C_1 (triangle)	50	0	1
C_2 (square)	0	36	0
C_3 (circle)	0	14	49

Conclusions

- Data is not a problem anymore, analyzing and understanding them still is.
- Data mining techniques may be organized into paradigms according to their principles, that is, a single paradigm may support several techniques.
- Paradigms are complementary and may be combined for a single technique.
- Understanding the paradigms and their characteristics is key to mine data effectively.
- Understanding the paradigms may also help significantly regarding grasping, exploiting and developing new techniques.

Four Paradigms in Data Mining

dataminingbook.info

Wagner Meira Jr.[1]

[1]Department of Computer Science
Universidade Federal de Minas Gerais, Belo Horizonte, Brazil

Thank you!

Questions?