CS 412 — Introduction to Machine Learning (UIC) April 12th, 2025

Lecture 22

Instructor: Aadirupa Saha *Scribe(s): Haoxuan Wang*

Overview

In the last lecture, we covered the following main topics:

1. K-Means clustering
2. Spectral clustering
3. Kernelized clustering

This lecture focuses on:

1. More discussion on spectral clustering
2. Basics of Neural Nets

1 Spectral Clustering

1.1 Some Preliminaries on Graph Cuts

Balanced Graph-Cutting

The goal is to cut a given graph $\mathcal{G}(V, E, W)$ into two sets A and B such that:

- The weight of edges connecting vertices in A to those in B is minimized.
- The sizes of A and B are "quite similar" (i.e., balanced).

Graph Definition

Assume any graph $\mathcal{G} = (V, E, W)$ where:

- V: set of vertices
- E: set of edges
- W: set of edge weights

For any two vertices $i, j \in V$, define:

$$e_{ij} = \mathbf{1}(i, j \text{ are connected}), \quad w_{ij} = \text{weight on edge } (i, j).$$

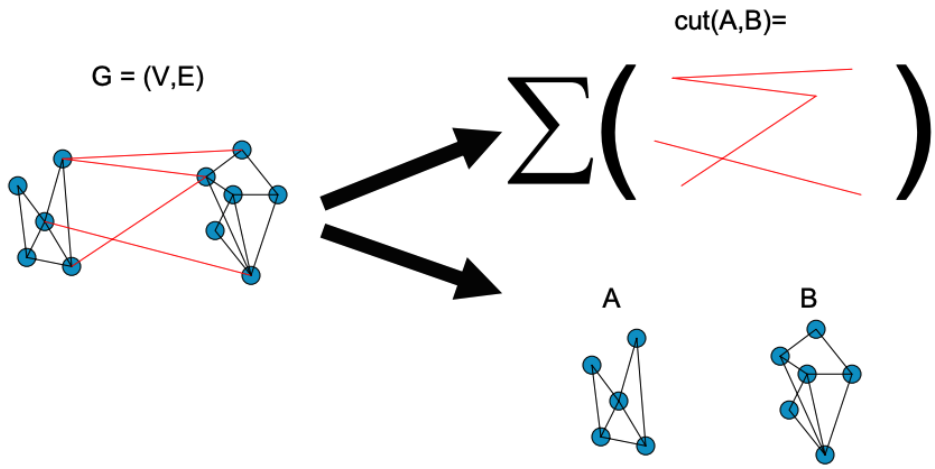

Figure 1: Example of a graph cut (Ref 2).

Graph Cut Definition

The weight of the cut between sets A and B is defined as:

$$\text{Graph}(A, B) := \sum_{i \in A, j \in B} w_{ij}.$$

This sum includes all edges crossing from set A to set B.

1.2 Formulations of the Graph Cut Problem

The problem can be formulated in several ways:

1. **Balanced Cut:**

$$\min_{A,B} \text{Cut}(A, B) \quad \text{s.t.} \quad |A| \approx |B|$$

2. **Ratio Cut:**

$$\min_{A,B} \text{Cut}(A, B) \left(\frac{1}{|A|} + \frac{1}{|B|} \right)$$

3. **Normalized Cut:**

$$\min_{A,B} \text{Cut}(A, B) \left(\frac{1}{\text{Vol}(A)} + \frac{1}{\text{Vol}(B)} \right),$$

   where:

$$\text{Vol}(A) = \sum_{i \in A} d_i, \quad \text{and} \quad d_i = \sum_{j:(i,j) \in E} w_{ij} \quad \text{(degree of vertex } i\text{)}$$

Quadratic Form Representation of Cuts

To solve the above problems, define a vector f corresponding to the partition A, B, such that:

$$f = (f_1, \dots, f_n) \in \{-1, 1\}^n, \quad \text{where } n = |V|$$

$$f_i = \begin{cases} 1, & \text{if } i \in \text{Partition A} \\ -1, & \text{if } i \in \text{Partition B} \end{cases}$$

Then the cut can be rewritten as:

$$\begin{aligned} \text{Cut}(A, B) &= \sum_{i \in A, j \in B} w_{ij} \\ &= \frac{1}{4} \sum_{i,j} w_{ij} (f_i - f_j)^2 \\ &= \frac{1}{2} f^\top (D - W) f \end{aligned}$$

1.3 Relaxing the Balanced Graph Cut Problem

The balanced graph cut problem can be rewritten as the following discrete optimization problem:

Problem P_1:

$$\min_{f \in \{-1,1\}^n} \quad f^\top L f \quad \text{s.t.} \quad f^\top \mathbf{1} = 0, \ f^\top f = n$$

(The constraint $f^\top \mathbf{1} = 0$ enforces balance: $\sum_{i \in A} f_i = -\sum_{j \in B} f_j = 0$)

But this formulation is **NP-hard**. So, we consider a relaxation:

Problem P_2:

$$\min_{f \in \mathbb{R}^n} \quad \frac{f^\top L f}{f^\top f} \quad \text{s.t.} \quad f^\top \mathbf{1} = 0$$

Approximation Guarantee of P_2

Since P_2 is an approximation of P_1, what can we say about its guarantees? Can we say that a solution to P_2 is also "nearly good" for P_1?

Theorem 1.1: Cheeger's Inequality

If $\mathcal{G}$ is an undirected, regular graph (i.e., each vertex has the same degree d), then:

$$\frac{\lambda_2}{2} \leq \min_{A \subseteq V} \frac{\text{Cut}(A, V \setminus A)}{\min(|A|, |V \setminus A|)} \leq \sqrt{2\lambda_2}$$

where λ_2 is the second smallest eigenvalue of the normalized Laplacian:

$$L = D^{-1/2}(D - A)D^{-1/2}$$

Moreover, a simple sorting-based algorithm can sort the eigenvector v_2 (corresponding to λ_2) to find a partition $A \subseteq V$ which is a $\sqrt{2\lambda_2}$-approximate solution to the original balanced cut problem P_1.

1.4 Properties of the Laplacian Matrix

- **Laplacian** $L = D - W$ is always *positive semidefinite* (PSD).

$$0 \;\leq\; \lambda_1 \;\leq\; \lambda_2 \;\leq\; \cdots \quad \Longrightarrow \quad \text{All eigenvalues of } L \text{ are nonnegative.}$$

- The smallest eigenvalue λ_1 is always 0:

$$\lambda_1 = 0.$$

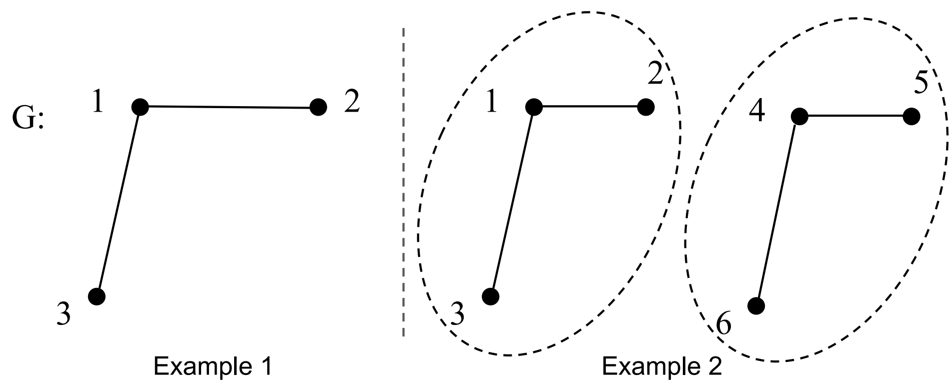

Figure 2: Examples of graph structures (Left: Example 1; Right: Example 2).

Example 1: Graph G with 3 Nodes

We have three nodes: 1, 2, and 3 (Fig. 2). Edges connect node 1 to node 2, and node 1 to node 3. Hence the adjacency matrix W (with $W_{ij} = 1$ if nodes i and j share an edge, and 0 otherwise) is:

$$W = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}.$$

The *degree matrix* D is diagonal, where D_{ii} is the degree of node i (the sum of the entries in row i of W):

$$D = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

The *graph Laplacian* is defined as $L = D - W$. Substituting the matrices above, we get:

$$L = D - W = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} - \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 2 & -1 & -1 \\ -1 & 1 & 0 \\ -1 & 0 & 1 \end{bmatrix}.$$

Suppose L is such that v_1 (often the constant vector) is an eigenvector for the eigenvalue 0. In a simplified form,

$$L\,v_1 \;=\; \begin{bmatrix} 2 & -1 & \cdots \\ -1 & 2 & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ \vdots \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \end{bmatrix}.$$

This shows v_1 is an eigenvector with eigenvalue 0, consistent with the property that L has at least one zero eigenvalue.

Example 2: Two Disconnected Components

Consider a graph with nodes $\{1, 2, 3\}$ forming one connected component and nodes $\{4, 5, 6\}$ forming another, each shaped like a simple chain (Fig. 2). Because there are no edges between these two sets, the Laplacian matrix L is block-diagonal:

$$L = \begin{bmatrix} 1 & -1 & 0 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & 0 & -1 & 1 \end{bmatrix}.$$

Because this graph has at least two disconnected components, the Laplacian L will have multiple zero eigenvalues. In the case of three disconnected components, for instance:

$$\lambda_1 = 0, \quad \lambda_2 = 0, \quad \lambda_3 = 0,$$

with corresponding eigenvectors

$$v_1 = [1, 1, 1, 1, 1, 1], \quad v_2 = [1, 1, 1, 0, 0, 0], \quad v_3 = [0, 0, 0, 1, 1, 1].$$

Each v_i is constant (i.e., has the same value) on one of the disconnected components, reflecting the fact that L has a separate zero eigenvalue for each component.

1.5 Spectral Embedding

Eigenvalues and Eigenvectors

From the graph Laplacian L, we obtain a set of eigenvalues and corresponding eigenvectors:

$$0 = \lambda_1 \ \leq \ \lambda_2 \ \leq \ \lambda_3 \ \leq \ \cdots \ \leq \ \lambda_d.$$

Each λ_i has an associated eigenvector v_i.

Case: $K = 2$ **Components**

After applying spectral embedding with $k = 2$:

$$\widetilde{D} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \in \mathbb{R}^{n \times 2},$$

where $x_i = (v_2(i), v_3(i))$ are the entries from the second and third smallest eigenvectors of the Laplacian matrix.

$$\text{If } x_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \ x_2 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \ x_3 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \text{ then assign } x_1, x_2 \text{ to cluster 1, and } x_3 \text{ to cluster 2.}$$

General K**-Dimensional Embedding**

For a general K-way spectral clustering, Given eigenvalues:

$$0 = \lambda_1 \ll \lambda_2 \leq \lambda_3 \leq \cdots \leq \lambda_d \quad \text{with eigenvectors} \quad v_1, v_2, \ldots, v_d,$$

we take:

$$\widetilde{D} \ = \ \begin{bmatrix} v_2 & v_3 & \cdots & v_{K+1} \end{bmatrix} \ \in \ \mathbb{R}^{n \times K}.$$

This serves as a **new representation of the data** in $\mathbb{R}^{n \times K}$, where $v_2, \ldots, v_{K+1}$ correspond to the K smallest *nonzero* eigenvalues of L. Each data point $x^{(i)}$ becomes a row in $\widetilde{D}$, and then we can apply a standard clustering method (e.g. K-Means) in this new K-dimensional space. This corresponds to finding a sparse graph cut in the original graph, based on the eigenstructure of L.

Final Step:

Apply k-means on $\widetilde{D}$ to identify the k clusters.

Remark 1. *Reflect on these examples in the* sparsest graph cut optimization view *discussed above.*

2 Neural Networks

In this lecture, we treat Neural Networks (NN) as a supervised learning framework for introduction.

2.1 Starting from Supervised Learning

We consider a dataset

$$D = \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(n)}, y^{(n)})\},$$

where each $x^{(i)} \in \mathbb{R}^d$ is a feature vector (an *instance*), and $y^{(i)}$ is a label, e.g. $y^{(i)} \in \{0, 1\}$ for binary classification.

Classical Methods

Examples of supervised learning algorithms include:

- **Logistic Regression (LR)**
- **Support Vector Machines (SVM)**

These methods learn a *predictor* that maps x to the label y.

Recall Logistic Regression: In a simple 2D feature space, the logistic regression *predictor* attempts to separate labeled points with a linear boundary. For a more complicated space, we can adopt a kernelized version to seperate data points.
However, there is a **limitation** on knowing which kernel to use:

- **Need to know which kernel** k to use. Formally, the kernel corresponds to an implicit feature mapping φ.

$$k(x, x') \quad \longleftrightarrow \quad \langle \varphi(x), \, \varphi(x') \rangle.$$

If we lack domain knowledge to select k, how can we *achieve* φ, thereby obviating the need to hand-craft the kernel?
Neural networks provide a more flexible framework for supervised learning. Like LR or SVM, they map x to y, but use multiple layers of nonlinear transformations to capture complex decision boundaries.
Note: A neural network (NN) tries to learn φ (the embedding) by training the parameters of the NN (instead of fixing φ ahead of time).

2.2 A Simple Single-Neuron Architecture

Let $x \in \mathbb{R}^d$ be an input vector with components $\{x_1, x_2, \ldots, x_d\}$. A basic neural network "neuron" can be described by:

$$z = \sum_{i=1}^{d} w_i \, x_i \; + \; b,$$

where w_i are the learned weights, and b is a bias term. We then apply an activation function σ, often the *sigmoid* function:

$$\sigma(z) = \frac{1}{1 + e^{-z}},$$

to obtain the output $f(x)$:

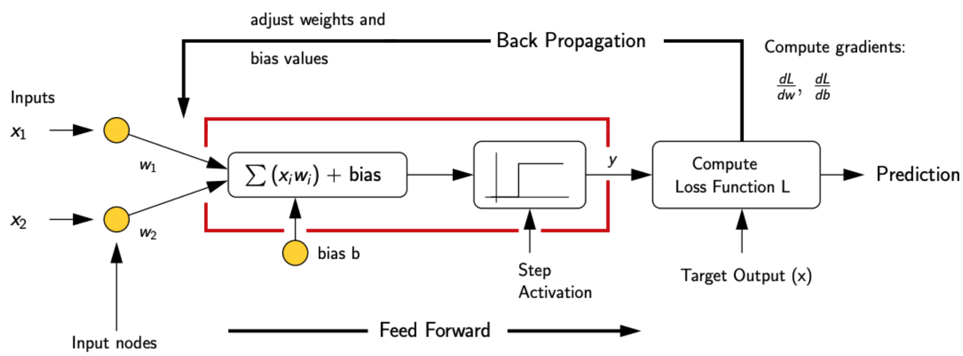

Figure 3: An illustration of a perceptron (Ref 2).

$$f(x) = \sigma\Big(\sum_{i=1}^{d} w_i\,x_i + b\Big).$$

This output $f(x)$ lies in the interval $(0,1)$ if σ is the sigmoid. Conceptually, each x_i connects to the neuron input with weight w_i, and the neuron's sum is offset by b. The illustration can be summarized as follows:

$$x_1,\, x_2,\, \dots,\, x_d \;\xrightarrow[\text{inputs}]{}\; \big(\text{linear sum: } z\big) \;\xrightarrow[\text{bias } b]{}\; \xrightarrow[\text{activation } \sigma]{} \; f(x).$$

2.3 One-Layer Neural Network

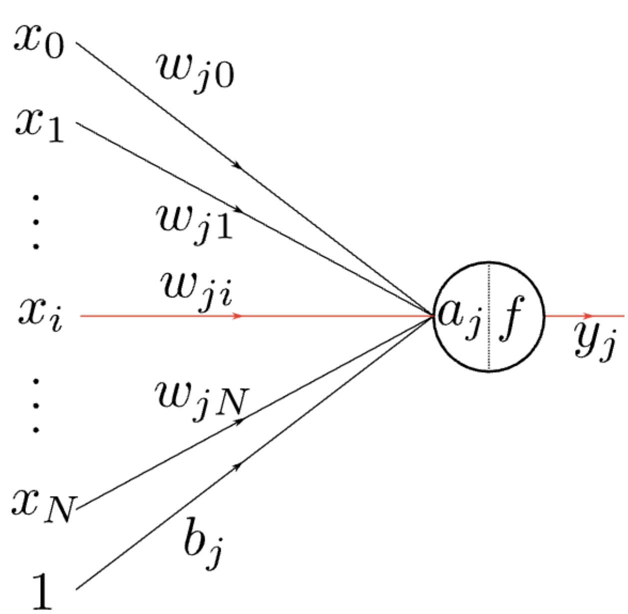

Figure 4: An illustration of a one layer NN. Ref 5.

Consider a neural network with input vector

$$(x_0, x_1, x_2, \ldots, x_N, 1)^\top \quad \text{where } x_{N+1} = 1 \text{ is the bias term,}$$

and $K = 1$ output neurons. Each output neuron k ($k = 1, \ldots, K$) computes an activation

$$y_j \;=\; f\Bigl(\sum_{i=0}^{N+1} w_i^{(k)}\, x_i\Bigr),$$

where:

- $w_i^{(k)}$ is the weight from input x_i to output neuron k.
- $f(\cdot)$ is an activation function, often a sigmoid or ReLU. In the figure, $f(z) = \frac{1}{1+e^{-z}}$, which maps real numbers to $(0, 1)$.

Illustration:

$$\underbrace{x_0 = 1,\, x_1,\, x_2,\, \ldots,\, x_d}_{\text{inputs}} \quad \longrightarrow \quad \sum_{i=0}^{d} w_i^{(1)} x_i \;\xrightarrow{f}\; y_1,$$

$$\sum_{i=0}^{d} w_i^{(2)} x_i \;\xrightarrow{f}\; y_2, \quad \ldots \quad \sum_{i=0}^{d} w_i^{(K)} x_i \;\xrightarrow{f}\; y_K.$$

Interpretation

- Each x_i is connected to every output neuron k with a weight $w_i^{(k)}$.
- The bias input $x_{N+1} = 1$ ensures each output neuron can learn an offset.
- The activation function σ is applied to the linear sum, creating a non-linear mapping from inputs to outputs $\{y_k\}$.

2.4 Types of Activation Functions

1) Sigmoid

The *sigmoid* (logistic) function maps $\mathbb{R}$ to the interval $(0, 1)$. For $w \in \mathbb{R}$,

$$\sigma(w) \;=\; \frac{1}{1 + e^{-w}}.$$

2) Tanh (Hyperbolic Tangent)

The *tanh* function maps $\mathbb{R}$ to the interval $[-1, 1]$. For $w \in \mathbb{R}$,

$$\tanh(w) \;=\; \frac{e^{2w} - 1}{e^{2w} + 1},$$

which takes values in $[-1, 1]$.

3) ReLU (Rectified Linear Unit)

The *ReLU* activation function maps $\mathbb{R}$ to $[0, \infty)$. For any real input w,

$$\text{ReLU}(w) = \max\{\, 0,\; w\}.$$

2.5 How to learn NN parameters

Parameter Tuning in Logistic Regression (LR)

Recall the logistic model:

$$f(x_i) = \frac{1}{1 + e^{-w^\top x_i}}$$

Log-Likelihood:

$$\log L(\mathcal{D}) = \sum_{i=1}^{n} \left[y_i \log\left(\frac{1}{1 + e^{-w^\top x_i}}\right) + (1 - y_i)\log\left(\frac{e^{-w^\top x_i}}{1 + e^{-w^\top x_i}}\right)\right]$$

Loss Minimization View:

Minimize the negative log-likelihood:

$$\arg\min_{w \in \mathbb{R}^{d+1}} \; -\log L(\mathcal{D})$$

Final Form: (highlighted)

$$\arg\min_{w \in \mathbb{R}^{d+1}} \sum_{i=1}^{n} \left[y_i \log\left(1 + e^{-w^\top x_i}\right) + (1 - y_i)\log\left(1 + e^{w^\top x_i}\right)\right]$$

Loss function:

$$\ell(y, \hat{y}) \to \mathbb{R}, \quad \text{e.g.} \quad \ell(y_i, \hat{y}_i) = y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)$$

Neural Networks: Same Idea for Parameter Tuning

For neural networks, parameters θ include weights from multiple layers:

$$\theta = \left(W_1^{(1)}, W_2^{(1)}, \ldots, W_k^{(1)}, W^{(2)}\right) \in \mathbb{R}^d$$

The network has $(d+1)k + 2$ parameters to tune, and the same optimization framework applies.

Likelihood Objective

Given dataset $\mathcal{D}$, the likelihood function is:

$$\mathcal{L}(\mathcal{D}) = \prod_{i=1}^{n} f(x_i)^{\mathbb{1}(y_i=1)}(1 - f(x_i))^{\mathbb{1}(y_i=0)}$$

Neural Network Model

$$f(x_i) = \text{NN}\left(x_i; w_1^{(1)}, w_2^{(1)}, \ldots, w_k^{(1)}, w^{(2)}\right)$$

Let θ be the collection of all neural network parameters. Then the prediction function becomes:

$$f_{\text{NN}}(x_i) = \text{NN}(x_i; \theta), \quad \theta \in \Theta$$

Loss Function

Define a general loss function:

$$\ell : \mathcal{Y} \times \hat{\mathcal{Y}} \to \mathbb{R}$$

Minimize training loss over data:

$$\arg\min_{\theta \in \Theta} \sum_{i=1}^{n} \ell\left(y_i, f_{\text{NN}}(x_i)\right)$$

This represents the total training loss on $\mathcal{D}$.

Optimization Strategy

Although ℓ is no longer convex in θ, even for simple loss functions like log-loss (cross-entropy loss), we still solve for θ using gradient descent (GD).

2.6 Training Neural Networks: Gradient Descent and Backpropagation

Gradient Descent (GD)

Let θ_0 be the initial estimate of parameters. For $t = 1, 2, \ldots$:

$$\theta_{t+1} \leftarrow \theta_t - \eta \nabla_\theta \mathcal{L}(\mathcal{D}; \theta_t)$$

This is standard Gradient Descent (GD) applied on θ using training loss $\mathcal{L}(\mathcal{D}; \theta)$.

Variants of GD

We can also use other variants of GD to improve computational efficiency:

1. **Stochastic Gradient Descent (SGD)**
2. **Mini-batch SGD**

Challenge: Computing Gradients

Computing $\nabla_\theta \mathcal{L}(\mathcal{D}; \theta)$ is **hard**, since $f_{\text{NN}}(\theta)$ is a *complicated* function.

Solution: Backpropagation

The solution is known as **Backpropagation** — a fancier name for the *chain rule of differentiation*.

Next Lecture

The next lecture will cover the following topics:
(i) Backpropagation.
(ii) Forward Propagation.
(iii) Regularization in NN.

References:

1. Lecture note by Ethan Fetaya, James Lucas and Emad Andrews from course CSC 411. Source
2. An Introduction to Graph-Cut by Paul Scovanner. Link
3. Lecture note by Mark A. Austin. Source
4. Blog on activation functions. Link
5. Blog on backpropagation. Link
6. ChatGPT, OpenAI