CS5228 Knowledge Discovery and Data Mining

AY2024/25 Sem2 By Zhao Peiduo

Lecture 1

Introduction

Common Data Mining Tasks Data mining encompasses various techniques to analyze and extract patterns from large datasets. Some of the most common data mining tasks include:

- **Association Rules:** This method analyzes *transactional data*, where a transaction is a data record consisting of a set of items from a fixed collection. The goal is to identify *association rules* that predict the occurrence of items based on the presence of other items in the dataset.
- **Clustering:** Clustering involves grouping data points based on a well-defined notion of similarity. The objective is to form *clusters*, ensuring that data points within the same cluster have high intra-cluster similarity while minimizing inter-cluster similarity with other clusters.
- **Classification:** This method uses datasets with multiple attributes to determine the *categorical value* of an attribute as a function of other attribute values. Popular classification techniques include K-Nearest Neighbor, Decision Trees, and Linear Classification.
- **Regression:** Similar to classification, regression also works with datasets having multiple attributes, but it predicts *numerical values* of an attribute as a function of other attributes. Common regression methods include K-Nearest Neighbor, Regression Trees, and Linear Regression.
- **Graph Mining:** This technique analyzes data represented as a graph, $G = (V, E)$, where V represents data points (vertices) and E represents relationships between them (edges). Typical patterns derived from graph mining include identifying communities of nodes and detecting important nodes within the network.
- **Recommender Systems:** Recommender systems work with *user-rated items* (such as movie ratings) to predict missing values and recommend items based on similarities. They exploit item features and user similarities to enhance recommendations.

Types of Attributes

- **Categorical (Qualitative):**
  - **Nominal:**
    - Values are only labels.
    - Operations: $=, \neq$
    - Examples: sex (m/f), eye color, zip code.
  - **Ordinal:**
    - Values are labels with a meaningful order.
    - Operations: $=, \neq, <, >$
    - Examples: street numbers, education level.
- **Numerical (Quantitative):**
  - **Interval:**
    - Values are measurements with a meaningful distance.
    - Operations: $=, \neq, <, >, +, -$
    - Examples: body temperature in °C, calendar dates.
  - **Ratio:**
    - Values are measurements with a meaningful ratio.
    - Operations: $=, \neq, <, >, +, -, \times, \div$
    - Examples: age, weight, income, blood pressure.

Types of Data

Data can be classified into three main types based on structure and organization:

- **(Well-)Structured Data:**
  - Highly organized: adheres to a predefined data model.
  - Each object has the same fixed set of attributes.
  - Easy to search, aggregate, manipulate, and analyze.
  - Examples: relational databases, spreadsheets.
- **Semi-Structured Data:**
  - No rigid data model: mix of structured and unstructured data.
  - Data exchange formats: XML, JSON, CSV.
  - Tagged unstructured data (e.g., photo with date/time, location, exposure, resolution, flash, etc.).
- **Unstructured Data:**
  - No fixed data model.
  - Requires more advanced data analysis techniques.
  - Examples: images, videos, audio, text, social media.

Data Quality

- **Noise**: Data can be defined as: true signal + **noise**. Sources of noise include:
  - Sensor readings from faulty devices (e.g., intrinsic noise or external influences).
  - Errors during data entry (by humans or machines).
  - Errors during data transmission.
  - Inconsistencies in data formats (e.g., ISO time vs. Unix time, DD/MM/YYYY vs. MM/DD/YYYY).
  - Inconsistencies in conventions (e.g., meters vs. miles, meters vs. centimeters).
- **Outliers**:An outlier is a data point with attribute values considerably different from other points.Outliers can be classified into:
  - **Outliers as noise:**
    - They negatively interfere with data analysis.
    - Removal of outliers or using robust methods is recommended.
  - **Outliers as targets:**
    - The goal is to detect rare or anomalous events such as credit card fraud detection and intrusion detection in security systems.
- **Missing Values**
  - Common causes of missing values:
    - Attribute values not collected (e.g., broken sensor, person refused to report age).
    - Attributes not applicable in all cases (e.g., no income data for children).
  - Handling missing values:
    - Remove data points with missing values.
    - Remove attributes with missing values (if not essential).
    - Try to fill in missing values (e.g., using average temperature from nearby sensors).
- **Duplicates**
  - Duplicates refer to data points representing the same object/entity.
    - **Exact duplicates:** Data points have identical attribute values.
    - **Near duplicates:** Data points slightly differ in their attribute values (e.g., same person with phone numbers in different formats).
  - Duplicate elimination:
    - Relatively easy for exact duplicates.
    - Challenging for near duplicates.

Exploratory Data Analysis (EDA)

Exploratory Data Analysis (EDA) is an essential step in data analysis to identify potential issues such as noise, outliers, missing values, and class distribution imbalances.

- **Identifying Noise**
  - Using histograms to inspect the distribution of data values.
- **Identifying Noise / Outliers**
  - Using box plots to inspect the distribution of attribute values.
    - Make outliers explicit.
  - Using scatter plots to inspect correlations.
    - Not always feasible in practice.
    - Requires good understanding of data.
- **Handling Missing Values**
  - Example: Default value (0) if people did not disclose weight.
    - Can negatively affect simple analysis such as calculating means/averages.
- **Distribution of Class Labels**
  - Classification tasks generally benefit from balanced datasets.
    - Balanced = all classes are (almost) equally represented.
    - Distribution of classes also affects the evaluation of found patterns.

Data Preprocessing

- **Main Purposes**
  - Improve data quality (*"Garbage in, garbage out!"*).
  - Generate valid input for data mining algorithms.
  - Remove complexity from data to ease analysis.
- **Core Preprocessing Tasks**
  - Data cleaning
  - Data reduction
  - Data transformation
  - Data discretization
- **Data Cleaning**
  - Remove or fill missing values.
  - Identify and remove outliers (if outliers are not the goal of the analysis).
  - Identify and remove/merge duplicates.
  - Correct errors and inconsistencies (e.g., convert inches to centimeters).
  - *Non-trivial tasks that are typically very application-specific.*
- **Data Reduction**
  - **Reducing the number of data points**
    - Sampling — selecting a subset of data points (random or stratified sampling).
    - Used for preliminary analysis or large datasets.
  - **Reducing the number of attributes**
    - Removing irrelevant attributes (e.g., IDs, sensitive attributes).
    - Dimensionality reduction (PCA, LDA, t-SNE).
  - **Reducing the number of attribute values**
    - Aggregation or generalization.
    - Binning with smoothing.
- **Data Transformation**
  - Some data reduction techniques also transform data (e.g., dimensionality reduction, aggregation, binning).
  - Attribute construction:
    - Add or replace attributes inferred from existing attributes.
    - Example: weight, volume $\rightarrow$ density.
  - Normalization:
    - Scaling attribute values to a specified range (e.g., [0,1]).
    - Standardization: scaling using mean and standard deviation.
- **Data Discretization**
  - Converting continuous attributes into ordinal attributes.
  - Some algorithms accept only categorical attributes.
  - Convert a regression task to a classification task.
- **One-Hot Encoding**
  - Converting categorical attributes into numerical attributes.
  - Transform categorical attributes into binary attributes (0/1).
  - Allows the application of numerical methods on categorical attributes.

Lecture 2

Clustering

Goal of Clustering

- Separate **unlabeled** data into groups of **similar** objects/points
- Maximize **intra-cluster** similarity
- Minimize **inter-cluster** similarity

Meso-level perspective on data

- **micro**: individual data points
- **meso**: clusters
- **macro**: whole dataset

Ingredients for Clustering

- **Representation of objects, e.g.:**
  - (Multidimensional) point coordinates x, y
  - Sets A, B (e.g., items in a transaction)
  - Vectors u, v (e.g., TF-IDF)
- **Similarity Measure, e.g.:**
  - Euclidean Distance: $dist_{\text{euclidean}}(x, y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$
  - Jaccard Similarity: $sim_{\text{jaccard}}(A, B) = \frac{|A \cap B|}{|A \cup B|}$
  - Cosine Similarity: $sim_{\text{cosine}}(u, v) = \frac{u \cdot v}{\|u\|\|v\|}$
- **Clustering Algorithm**
  - Process that determines if an object belongs to a cluster

Types of Clusters

- **Well-separated vs. Center-based**
  - Well-separated: Any object in a cluster is closer to every other object in the cluster than to any point outside the cluster.
  - Center-based: Any object in a cluster is closer to the "center"of the cluster than to the center of any other cluster. The cluster center is commonly called a **centroid**.
- **Contiguity-based vs. Density-based**
  - Contiguity-based: Two objects are in the same cluster if they are more similar than a specified threshold, ensuring each object is more similar to some object in the cluster than to any point in a different cluster.
  - Density-based: A cluster is defined as a dense region of objects surrounded by regions of lower density, which allows better handling of noise.
- **Partitional vs. Hierarchical**
  - Partitional: The data is divided into non-overlapping subsets (i.e., clusters), where each object belongs to exactly one cluster or no cluster at all.
  - Hierarchical: Clusters can be nested, and a point can belong to different clusters depending on the hierarchy level.
- **Exclusive vs. Non-exclusive / Overlapping**
  - Exclusive: Each object belongs to exactly one cluster.
  - Non-exclusive / Overlapping: An object can belong to more than one cluster at a time. Fuzzy clustering assigns each object to all clusters with a certain probability.
- **Complete vs. Partial**
  - Complete: Every object is assigned to at least one cluster, ensuring full coverage of data points.
  - Partial: An object might not belong to any cluster, allowing for the presence of noise and outliers.

K-Means

- **Basic Characteristics**
  - Clusters are centroid-based.
  - Clustering is partitional, exclusive, and complete.
- **Inputs (for d-dimensional Euclidean space)**
  - Data points: $(x_1, x_2, ..., x_N), \quad x_i \in R^d$
  - Number of clusters: $K \to C_1, C_2, ..., C_K$ (cluster centers).
- **Optimization Objective**
  - Minimize Sum of Squared Error (SSE): $SSE = \sum_{i=1}^{K} \sum_{x \in C_i} \|x - c_i\|^2$
  - Finding the optimal solution is NP-hard: $O(N^{Kd+1})$.

K-Means — How to Define the Centroid of a Cluster?

Simple case in Euclidean space

The centroid is derived by minimizing SSE, which turns out to be the mean of all points in that cluster. Proof:

$$
\begin{aligned}
SSE &= \sum_{i=1}^{K} \sum_{x \in C_i} \|x - c_i\|^2 \\
\frac{\delta}{\delta c_k} SSE &= \frac{\delta}{\delta c_k} \sum_{i=1}^{K} \sum_{x \in C_i} \|x - c_i\|^2 \\
&= \sum_{x \in C_k} \frac{\delta}{\delta c_k} \|x - c_k\|^2 \\
&= \sum_{x \in C_k} 2(x - c_k)
\end{aligned}
$$

$$
\begin{aligned}
\sum_{x \in C_k} 2(x - c_k) &= 0 \\
\sum_{x \in C_k} x - \sum_{x \in C_k} c_k &= 0 \\
m_k c_k &= \sum_{x \in C_k} x \\
c_k &= \frac{1}{m_k} \sum_{x \in C_k} x
\end{aligned}
$$

K-Means — Basic Algorithm (Lloyd's Algorithm)

- **Initialization**
  - Select K points as initial centroids: $C_1, C_2, ..., C_K$.
- **Repeat**
  - **Assignment**: Assign each point to the nearest cluster (i.e., centroid).
  - **Update**: Move each centroid to the average of its assigned points.
- **Stopping Criterion**
  - Repeat until no change in assignments.

Handling Empty Clusters

- **Artificially fill empty clusters after assignment step**
  - Replace empty cluster with the point that contributes most to SSE.
  - Replace empty cluster with a point from the cluster with the highest SSE.
- **Postprocessing**
  - Split "loose"clusters with very high SSE.
- **Modification of Lloyd's Algorithm**
  - K-Means variants aim to address the initial centroids issue.

K-Means — Limitations and Potential Workarounds

- **Susceptibility to "Natural"Clusters**
  - K-Means struggles with:
    - Non-spherical clusters.
    - Clusters of different sizes.
    - Clusters of different densities.
  - **Potential Workaround: Choose a Larger Value for K**
    - Split natural clusters into multiple "well-behaved"subclusters.
    - Apply suitable postprocessing steps to merge subclusters.
- **Sensitivity to Initial Centroid Selection**
  - Different initializations of centroids may yield:
    - Different clusterings with varying SSEs (leading to local optima).
    - Empty clusters if a centroid is "blocked off"by other centroids.
  - **Potential Workaround: Improve Centroid Initialization**
    - **Artificially Fill Empty Clusters After Assignment**
      - Replace empty cluster with the point that contributes most to SSE.
      - Replace empty cluster with a point from the cluster with the highest SSE.
    - **Postprocessing**
      - Split "loose"clusters with very high SSE.
    - **Modification of Lloyd's Algorithm (K-Means Variants)**
      - Use improved centroid initialization techniques.

K-Means Variants

- **K-Means++**
  - Only changes the initialization of centroids.
  - Goal: Spread out centroids for better performance with theoretical guarantees.
  - **Initialization Process:**
    - Pick a random point as the first centroid C_1.
    - Repeat:
      - For each point x, calculate distance d_x to the nearest existing centroid.
      - Pick a random point for the next centroid with probability proportional to d_x^2.
    - Until K centroids have been picked.
- **X-Means**
  - Automatic method to choose K.
  - Iteratively applies K-Means with $K = 2$ to refine clustering.
  - **Example Scoring Functions:**
    - Bayesian Information Criterion (BIC).
    - Akaike Information Criterion (AIC).
    - Minimum Description Length (MDL).
- **K-Medoids**
  - **Restriction:** Centroids are chosen from the data points.
    - Does not require calculation of averages.
    - Uses only a notion of distance or similarity.
    - More robust to noise and outliers.
  - **Main Issue: Performance**
    - More expensive update step.
    - Swap medoid with each point in the cluster and calculate change in cost (e.g., SSE).
    - Choose the point as the new medoid that minimizes cost after swapping.

DBSCAN (Density-Based Spatial Clustering of Applications with Noise)

Basic Characteristics

- **Clusters:** Density-based
- **Clustering:** Partitional, Exclusive, Partial

Inputs (for d-dimensional Euclidean space)

- $\mathbf{x} = (x_1, x_2, ..., x_N), x_i \in \mathbb{R}^d$
- ε (Epsilon): Radius defining a point's neighborhood
- $MinPts$: Minimum number of points in a neighborhood

$density = \frac{mass}{volume} = \frac{MinPts}{\varepsilon}$

Types of Points in DBSCAN

- **Core points**
  - Points with at least $MinPts$ neighbors within radius ε
  - Form the **interior** of a cluster
- **Border points**
  - Non-core points with at least one core point in their neighborhood
  - Form the **border** of a cluster
- **Outliers / Noise**
  - All other points
  - Default node type

Algorithm 1 DBSCAN Algorithm

```
1: Input: Dataset $D$, radius $\varepsilon$, minimum points $MinPts$
2: Output: Clusters and outliers
3: for each point $P$ in $D$ do
4:     if $P$ is already visited then
5:         Continue
6:     end if
7:     Mark $P$ as visited
8:     Retrieve all neighbors $N$ of $P$ within radius $\varepsilon$
9:     if $|N| < MinPts$ then
10:        Mark $P$ as noise
11:    else
12:        Create a new cluster
13:        Expand cluster by adding density-reachable points
14:    end if
15: end for
```

Characteristics of DBSCAN

- **DBSCAN always converges**
  - Every data point is explored in either Phase 1 or Phase 2
  - A data point does not change its type (exception: noise → border)
- **DBSCAN is not completely deterministic**
  - Phase 1 introduces randomness
  - Border points may be reachable from core points of different clusters
  - Noise and core points are deterministic

DBSCAN — Limitations

- DBSCAN cannot handle different densities.
- DBSCAN is generally very sensitive to parameters.Choosing ϵ and $MinPts$ requires a good understanding of data and context.

DBSCAN — How to Choose Parameter Values?

- Informed by results of EDA, e.g.:
  - Distribution of all pairwise distances.
  - First insights into suitable values for ϵ.
- Density of data points has intuitive semantic meaning, e.g.:
  - Geographic distance between bars in a city.
  - Task: Find areas (clusters) with more than 10 bars within 500m.

Lecture 3

Hierarchical Clustering

Basic characteristics
- Clusters: depends...
- Clustering: hierarchical, complete, exclusive (at each level!)

No parameterization (in principle)
- In practice, typically the number of clusters is specified (similar to K-means)
- Different choices of measures to calculate distances between clusters

Dendrograms
- A dendrogram is a visualization of hierarchical relationships
- Binary tree showing how clusters are hierarchically merged/split
- Each node represents a cluster
- Each leaf is a singleton cluster
- Height reflects distance between clusters

Two Main Types of Hierarchical Clustering
- **Agglomerative (bottom-up)**
  - Start with each point being its own cluster
  - Merge the closest pair of clusters at each step
  - Stop when only one cluster remains
  - Example: **AGNES (Agglomerative Nesting)**
- **Divisive (top-down)**
  - Start with all points in a single cluster
  - Recursively split clusters at each step
  - Stop when each cluster contains a single point
  - Example: **DIANA (Divisive Analysis)**

AGNES Algorithm

Algorithm 2 AGNES Algorithm

```
1: Input: Dataset $D$
2: Output: Hierarchical clusters
3: for each point $P$ in $D$ do
4:     Assign $P$ to its own cluster
5: end for
6: while more than one cluster exists do
7:     Merge the two closest clusters into one
8: end while
```

AGNES Algorithm Linkages
- **Single Linkage Clustering**
  - Distance between clusters = **minimum distance** between two points from each cluster: $d_{\text{single}}(C_i, C_j) = \min_{p\in C_i, q\in C_j} d(p,q)$
  - **Strength**: Can handle non-globular shapes.
  - **Weakness**: Very susceptible to noise (*Chaining Effect*).
- **Complete Linkage Clustering**
  - Distance between clusters = **maximum distance** between two points from each cluster: $d_{\text{complete}}(C_i, C_j) = \max_{p\in C_i, q\in C_j} d(p,q)$
  - **Strength**: Less susceptible to noise or outliers.
  - **Weakness**: Bias towards globular clusters, breaks large clusters.
- **Average Linkage Clustering**
  - Distance between clusters = **average distance** between two points from each cluster: $d_{\text{average}}(C_i, C_j) = \text{avg}_{p\in C_i, q\in C_j} d(p,q)$
- **Centroid Linkage**
  - Distance between clusters = distance between centroids: $d_{\text{centroid}}(C_i, C_j) = d(m_i, m_j)$
- **Ward Linkage**
  - Variance-based merging criterion:

$$d_{\text{ward}}(C_i, C_j) = \sum_{k\in C_i\cup C_j} \|x_k - m_{ij}\|^2 - \sum_{k\in C_i} \|x_k - m_i\|^2 - \sum_{k\in C_j} \|x_k - m_j\|^2$$

AGNES Complexity Analysis
- **Space Complexity**: $O(N^2)$ (storing distance matrix)
- **Time Complexity**:
  - Baseline: $O(N^3)$ - (N-1) steps, each step $O(N^2)$
  - Using heap/priority queue: $O(N^2 \log N)$
  - Single Linkage special optimization: $O(N^2)$

DIANA Algorithm

- **Top-Down Hierarchical Clustering**
  - Start with all points in a single cluster
  - Recursively split clusters until each point is its own cluster
- **Challenge:** 2^n ways to split a cluster with n points
  - Heuristics required to restrict the search space
  - Generally slower and less common than AGNES
- **Cases where DIANA is preferable:**
  - When no complete clustering is needed (early stopping)
  - When splitting can use global knowledge

Cluster Evaluation

Cluster Evaluation Problems
- Challenges in Visual Inspection: Just eyeballing the clustering is rarely possible.
- Algorithmic Bias: Clustering algorithms will always find some clusters.

Purpose of Cluster Evaluation
- Comparing the results of different clustering algorithms.
- Comparing the results of a clustering algorithm with different parameters.
- Minimizing the effects of noise on the clustering.

Two Main Approaches of Clustering Evaluation
- **External quality measures**: Evaluate clustering against a ground truth (if available).
- **Internal quality measures**: Evaluate clustering from the data itself.

External Quality Measures
- **Ground Truth: Labeled Data**
  - Labels indicate that two points "belong together."
  - If clustering reflects this, it is considered good clustering.
- **Cluster Purity**
  - Measures how pure clusters are concerning the most common label.
  - Formula: $P = \frac{1}{N}\sum_{c\in C} \max_{l\in L} |c \cap l|$
  - Example Calculation: $P = \frac{1}{8}(3+4) = 0.875$
  - **Limitation:** Purity does not penalize having many clusters.
  - $P = 1$ is easy to achieve if each cluster contains only a single point.
- **Information Retrieval Metrics**
  - **True Positives (TP)**: Same cluster, same label.
  - **True Negatives (TN)**: Different clusters, different labels.
  - **False Positives (FP)**: Same cluster, different labels.
  - **False Negatives (FN)**: Different clusters, same label.
- **Rand Index (RI)**
  - Reflects clustering accuracy.
  - Formula: $RI = \frac{TP+TN}{TP+FP+FN+TN}$
  - Example: $RI_{example} = 0.82$
- **Precision, Recall, and F1-Score**
  - Precision: $P = \frac{TP}{TP+FP}$
  - Recall: $R = \frac{TP}{TP+FN}$
  - F1-Score: $F1 = \frac{2\cdot P\cdot R}{P+R}$
  - Example values: $P_{example} = 0.75, \quad R_{example} = 0.69, \quad F1_{example} = 0.72$

Internal Quality Measures
- **Sum of Squared Errors (SSE)**
  - SSE is often used to determine the number of clusters.
  - Formula: $SSE = \sum_{i=1}^{k}\sum_{x\in C_i} ||x-\mu_i||^2$
  - **Limitations:**
    - Does not penalize having a large number of clusters.
    - SSE decreases as the number of clusters increases.
    - Applicable beyond K-Means but less intuitive for non-globular clusters.
  - SSE can also be used for complex data but inherently favors globular clusters.
- **Silhouette Coefficient**
  - A good clustering has:
    - High inter-cluster distances.
    - Low intra-cluster distances.
  - Defined for each data point x:
    - **Cohesion** $a(x)$: average distance to points in the same cluster.
    - **Separation** $b(x)$: minimum average distance to points in a different cluster.
    - **Silhouette Score**:

$$s(x) = \frac{b(x) - a(x)}{\max\{a(x), b(x)\}}, \quad \text{if } |C_x| > 1$$

    - If $|C_x| = 1$, then $s(x) = 0$.
  - The overall **Silhouette Coefficient** for the clustering:

$$SC = \frac{1}{N}\sum_{i=1}^{N} s(x_i)$$

  - Interpretation: $-1 \le s(x) \le 1$ where **negative values indicate poor clustering** and **values closer to 1 indicate well-separated clusters**.

Pragmatic Considerations
- The choice of the "best"clustering is often pragmatic:
  - Fixed number of clusters (problematic for DBSCAN).
  - Parameters defined by tasks (e.g., "areas with more than 5 McDonalds within 500m").
  - Maximum, minimum, or average size of clusters.
  - Focus on individual clusters instead of the whole clustering (e.g., largest/smallest cluster).
  - Setting k too high and merging later if needed.

Lecture 4

Association Rule Mining

Basic Setup
- **Input database:**
  - Set of **transactions**
  - Transaction = set of **items**
- **Output: Association Rules**
  - Rules predicting the occurrence of some items based on the occurrence of other items.

Example Association Rules

- **Antecedent → Consequent**
- $\{item_2, item_3\} \rightarrow \{item_5\}$
- $\{item_1\} \rightarrow \{item_3\}$

Problem Statement

- Association rules are not "hard" rules
  - Example: {cereal} → {milk} does not mean customers always buy milk with cereal.
  - Each possible combination (e.g., {yogurt, bread} → {milk}) is a potential rule.
- Given d unique items, the number of possible rules is: $3^d - 2^{d+1} + 1$
- Example: If $d = 6$, then there are 602 possible rules.

Applications of Association Rule Mining

- Market Basket Analysis: Understanding customer shopping behavior.
- Medical Data Analysis: Diagnosis Support Systems and ADR discovery (adverse drug reaction)
- Census Data Analysis: Getting insights into a population.
- Behavior Data Analysis: User preferences & linkings.

Association Rule Definitions

- **Itemset**: A subset of items.
- **K-itemset**: An itemset containing k items.
- **Support Count (SC)**: Number of transactions containing an itemset.
- **Support (S)**: Fraction of transactions containing an itemset.
- **Frequent Itemset**: An itemset with a support greater or equal to a minimum threshold $minsup$.
- **Association Rule**: An implication expression $X \rightarrow Y$, where X and Y are itemsets.
- **Support of an Association Rule**: Fraction of transactions containing all items in an association rule $X \rightarrow Y$:

$$S(X \rightarrow Y) = \frac{SC(X \cup Y)}{N} = S(X \cup Y)$$

- **Confidence of an Association Rule**: Probability of Y given X:

$$C(X \rightarrow Y) = \frac{S(X \rightarrow Y)}{S(X)} = \frac{S(X \cup Y)}{S(X)}$$

Support-Confidence Combinations

- **Low Support, Low Confidence**: Items in $(X \cup Y)$ do not frequently appear together. Even when items in X appear together, they often do so without items in Y.
- **High Support, Low Confidence**: Items in $(X \cup Y)$ frequently appear together, but if items in X appear together, they often do so without items in Y.
- **Low Support, High Confidence**: Items in $(X \cup Y)$ do not frequently appear together, but when items in X appear together, they often do so with items in Y.
- **High Support, High Confidence**: Items in $(X \cup Y)$ frequently appear together, and if items in X appear together, they often do so with items in Y.

Brute Force Algorithm

- **Finding Association Rules:** Given a set of transactions, find all association rules $X \rightarrow Y$ with:
  - Support $S(X \rightarrow Y) \geq minsup$
  - Confidence $C(X \rightarrow Y) \geq minconf$
- **Brute Force Algorithm:**
  - List all possible association rules $X \rightarrow Y$.
  - Calculate support $S(X \rightarrow Y)$ and confidence $C(X \rightarrow Y)$ for each rule.
  - Drop rules where $S(X \rightarrow Y) < minsup$ and $C(X \rightarrow Y) < minconf$.

Algorithm 3 Brute Force Association Rule Mining

```
1: Input: Transaction database $D$, minimum support $minsup$, minimum confidence $minconf$
2: Output: Set of association rules $R$
3: Initialize an empty set $F$ for frequent itemsets
4: Initialize an empty set $R$ for valid rules
                                                ▷ Frequent Itemset Generation
5: for each transaction $t$ in $D$ do
6:     for $k = 1$ to $|t|$ do
7:         Generate all possible $k$-itemsets
8:         for each itemset $I$ in $k$-itemsets do
9:             Count occurrences of $I$
10:        end for
11:    end for
12: end for
13: Retain only itemsets with support $\geq minsup$
                                                ▷ Rule Generation
14: for each frequent itemset do
15:     Generate all possible rules $X \rightarrow Y$
16:     Compute $S(X \rightarrow Y)$ and $C(X \rightarrow Y)$
17:     if $S(X \rightarrow Y) \geq minsup$ and $C(X \rightarrow Y) \geq minconf$ then
18:         Add $X \rightarrow Y$ to $R$
19:     end if
20: end for
21: return $R$
```

Computational Complexity:

- Given d unique items, the number of possible rules is:

$$3^d - 2^{d+1} + 1 \in O(3^d)$$

- Let w be the maximum number of items in a transaction within the database:

$$O\Big(N \cdot (3^w - 2^{w+1} + 1)\Big)$$

  where typically $w \ll d$.
- Frequent itemset generation complexity:

$$O(N \cdot 2^w)$$

Observation: Decoupling Support and Confidence:

- A rule $X \rightarrow Y$ has sufficient support only if $X \cup Y$ is a frequent itemset.
- No need to calculate confidence of rules where $X \cup Y$ is not a frequent itemset.

Apriori Algorithm

- **Apriori Principle (Anti-Monotonicity):**
  - If an itemset X is **not frequent**, then any superset $Y \supset X$ is **also not frequent**.
  - If an itemset Y is **frequent**, then all its subsets $X \subset Y$ must also be **frequent**.
- **Apriori Algorithm:**
  - Iteratively generate candidate frequent itemsets.
  - Use previously found frequent $(k-1)$-itemsets to **prune** candidate k-itemsets.
  - Filter frequent itemsets based on the minimum support threshold.

Algorithm 4 Apriori Algorithm for Frequent Itemset Mining

```
1: Input: Transaction database $D$, minimum support $minsup$
2: Output: Frequent itemsets $F$
3: Initialize $F_1$ with frequent 1-itemsets
                                                ▷ Frequent Itemset Generation
4: for $k = 2$ to max itemset length do
5:     Generate candidate $k$-itemsets $L_k$ from $F_{k-1}$
6:     Prune infrequent itemsets from $L_k$
7:     for each transaction $t$ in $D$ do
8:         for each candidate itemset $I$ in $L_k$ do
9:             Count occurrences of $I$
10:        end for
11:    end for
12:    Retain only itemsets with support $\geq minsup$ as $F_k$
13:    if $F_k$ is empty then
14:        break
15:    end if
16: end for
                                                ▷ Rule Generation
17: for each frequent itemset do
18:     Generate all possible rules $X \rightarrow Y$
19:     Compute $S(X \rightarrow Y)$ and $C(X \rightarrow Y)$
20:     if $C(X \rightarrow Y) \geq minconf$ then
21:         Add $X \rightarrow Y$ to $R$
22:     end if
23: end for
24: return $F$
```

- **Rule Generation:**
  - Generate association rules $X \rightarrow Y$ from frequent itemsets.
  - Compute confidence:

$$C(X \rightarrow Y) = \frac{S(X \cup Y)}{S(X)}$$

  - Retain rules where $C(X \rightarrow Y) \geq minconf$.
- **Computational Complexity:**
  - Generates at most **$2^d - 1$** itemsets for d unique items.
  - Reduces complexity by **pruning** infrequent itemsets early.
  - Worst-case complexity:

$$O(N \cdot 2^w)$$

  where N is the number of transactions and w is the largest transaction size.
- **Advantages:**
  - Reduces the number of candidate itemsets compared to brute-force.
  - Efficient for dense datasets with many frequent itemsets.
- **Limitations:**
  - Computationally expensive when itemsets are very large.
  - Many database scans required, increasing execution time.

Classification & Regression

Classification AND Regression:
- Core tasks of supervised machine learning.
- **Training:** Find patterns in a dataset between the values of **dependent variable(s)** given the values of **independent variable(s) / features**.
- **Prediction:** Use patterns to assign values to dependent variables for new/unseen data.

Classification VS. Regression:
- **Classification:** Dependent variable is categorical.
- **Regression:** Dependent variable is continuous.

Application Examples
- **Real Estate:** Prediction of flat prices (Regression task).
- **Health Analytics:** Prediction of heart disease (Classification task).
- **Text Analytics:** Sentiment Analysis / Opinion Mining (over text).
- **Self-Driving Vehicles:**
  - **Input:** Image & sensor data.
  - **Regression tasks, e.g.:**
    - Acceleration.
    - Steering angle.
    - Event prediction (%).
  - **Classification tasks, e.g.:**
    - Obstacle detection.
    - Street sign identification.

Supervised Learning

- **Basic Setup:**
  - Given: Dataset D of pairs $\{(x, y)\}$
    - x — features (independent variables).
    - y — label (dependent variable).
  - Split dataset D into:
    - D_{train} — data for training the model.
    - D_{test} — data for evaluating the model.
    - Important: $D_{train} \cap D_{test} = \emptyset$ (test data is not used during training).
- **Model:** Parameterized function $h(x, \Theta) = y$.
  - Maps input space (features) to output space (labels).
  - Θ — trainable/learnable parameters of the model.
- **Model Selection:**
  - Selection of a "family"of functions $h(x, \Theta) = y$.
  - Examples:
    - K-Nearest Neighbor.
    - Decision Trees.
    - Linear Models.
    - Neural Networks.
- **Training / Learning:**
  - Process of systematically finding the best values for Θ.
  - $\Theta_{best} \iff$ best mapping between features and labels.
- **Evaluation Process:**
  - Train model on (X_{train}, y_{train}).
  - Test model on unseen (X_{test}).
  - Generate predictions $\hat{y}_{test}$.
  - Compare $\hat{y}_{test}$ with ground truth y_{test}.

Classification and Regression Evaluation

- **Regression Evaluation**
  - Direct comparison of numerical values.
  - Common metric: Root Mean Squared Error (RMSE):

$$RMSE = \sqrt{\frac{\sum_{i=1}^{N}(\hat{y}_i - y_i)^2}{N}}$$

- **Classification Evaluation:** Comparison using a confusion matrix:
  - True Positives (TP): Correctly predicted positives.
  - True Negatives (TN): Correctly predicted negatives.
  - False Positives (FP): Incorrectly predicted positives.
  - False Negatives (FN): Incorrectly predicted negatives.

  Popular Classification Metrics

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN}$$

$$Sensitivity/Recall = \frac{TP}{TP + FN}$$

$$Specificity = \frac{TN}{TN + FP}$$

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

$$F1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$

Challenges in Evaluation
- **Imbalanced datasets:** Standard metrics may not be reliable
- **False Positives (FP) vs False Negatives (FN):** Different costs of errors
  - Example: **Heart disease prediction** (Recall > Precision)
  - Example: **News classification** (Precision > Recall)

Numerical Class Scores
- Many models output class scores instead of hard labels
- **Thresholding:** Convert scores to binary labels (e.g., threshold = 0.5)

Receiver Operating Characteristic (ROC) Curve
- Plots True Positive Rate (Sensitivity) vs. False Positive Rate (1-Specificity)
- Area Under ROC Curve (AUROC) quantifies classifier performance
  - AUROC = 0.5: Random classifier
  - AUROC = 1.0: Perfect classifier

One-vs-Rest: Macro vs. Micro Averaging
- **Micro-Averaging:** Averages over all individual predictions (favors larger classes)
- **Macro-Averaging:** Averages over metrics from each class equally

K-Nearest Neighbor Algorithm (KNN)

Intuition behind KNN:
- Label of an unseen data point x derives from the labels of the k-nearest neighbors of x.
- Similar data points → similar labels.
- **Required:** Notion of similarity/distance.

KNN for Classification
- **Training:** Remember training data.
- **Prediction for unseen point x_i**
  - Calculate distances to all training data points x_j, e.g.:

$$d(p, q) = \sqrt{\sum_{i}^{d}(q_i - p_i)^2} \tag{1}$$

  - Get the k-nearest neighbors.
  - **Label of x_i = most frequent label among all k-nearest neighbors.**

KNN for Classification — Distance Metrics
- **Examples for different distance metrics**
  - Euclidean distance: $d(p, q) = \sqrt{\sum_{i=1}^{d}(q_i - p_i)^2}$
  - Manhattan distance: $d(p, q) = \sum_{i=1}^{d} |q_i - p_i|$
  - Chebyshev distance: $d(p, q) = \max_i |q_i - p_i|$
- **Other metrics**
  - Cosine similarity: $\text{sim}(p, q) = \frac{\sum_{i=1}^{d} p_i q_i}{\|p\| \|q\|}$
  - Jaccard similarity: $\text{sim}(A, B) = \frac{|A \cap B|}{|A \cup B|}$

Common Outcomes
- Different k values yield different results:
  - **Very small** k: Sensitive to noise and outliers.
  - **Large** k: Insufficient capacity to properly separate.
  - **Very large** k: Most frequent class in training data dominates.

KNN for Regression
- **Training:** Remember training data.
- **Prediction for unseen point x_i**
  - Calculate distances to all training data points x_j, e.g.:

$$d(p, q) = \sqrt{\sum_{i}^{d}(q_i - p_i)^2} \tag{2}$$

  - Get the k-nearest neighbors.
  - **Label of x_i = mean of scores of all k-nearest neighbors.**

K-Fold Cross Validation & Model Evaluation

Extended Training Setup
- **Building a good classifier or regressor**
  - Find the best data preprocessing steps.
  - Find the best model (model selection).
  - Find the best hyperparameter values.
  - **Iterative evaluation required.**
- **Important: Not allowed to use test data for this!**
  - Test data is meant to be **unseen**.
  - Using test data during optimization makes it no longer unseen.

Training & Evaluation Using Validation Data
- **Common data split to ensure generalizability**
  - Use **test data** only at the end for final evaluation.
  - Use **validation data** for model selection and hyperparameter tuning.

K-Fold Cross Validation
- **Core idea**
  - Split training data into **k blocks** of equal size (e.g., $k = 10$).
  - Use **(k-1) blocks** for training and the remaining block for evaluation.
  - Repeat for **k rounds** with different permutations.
- **Advantages**
  - **Averaged results** are more reliable.
  - **Variance between results** is a useful indicator.

Information Leakage through Normalization
- **Important guidelines**
  - **Do not normalize before splitting** into training & test data!
  - **Normalize training & test data, but only based on training data!**

Decision Tree

Decision Tree — Idea
- Represent mapping between features and label/value as a flowchart-like structure.

Components
- (Inner) node — test on a single feature.
- Branch — outcome of a test; corresponds to a feature value or range of values.
- Leaf — label (classification) or real value (regression).

Same Dataset, Different Decision Tree
- In general, there are multiple trees that match a dataset.

Decision Tree — Diversity
- Different branching factors.
- Different depth:
  - Leaves can have more than one label or real value.
  - Required based on dataset or choice (pruning).
  - Final output: majority label (classification) or mean of values (regression).

Building a Decision Tree
- Notations:
  - D_t — set of records that reach node t.
  - D_0 — set of all records at root node.
- General procedure:
  - If $|D_t| = 1$ or all records in D_t have the same class or value $\Rightarrow$ t is a leaf node.
  - Otherwise, choose a test (feature + conditions) to split D_t into smaller subsets (subtrees).
  - Recursively apply the procedure to each subtree.

How to Split? — Nominal Attributes
- Binary split: Partition all d values into two subsets. $\frac{2^d-2}{2}$ possible splits.
- Multiway split: Each value yields a subtree. Arbitrary splits into $2 \leq s \leq d$ subtrees are possible, but the number of possible splits grows exponentially.

How to Split? — Ordinal Attributes
- Binary split: Partition all d values into two subsets. Partitions must preserve the natural order of values. $d-1$ possible splits.
- Multiway split: Each value yields a subtree.

Finding the Best Splits — Approaches
- Global Optimum:
  - Best splits = splits that result in a Decision Tree with the highest accuracy.
  - Problem: Finding the optimal tree is NP-complete $\Rightarrow$ not practical for large datasets.
- Greedy Approach:
  - Faster heuristics but no guarantees for optimal results.
  - Pick the split that minimizes the impurity of subtrees (w.r.t. class labels).

Finding the Best Splits — General Procedure
- Calculate impurity $I(t)$ of node t before splitting.
- For each possible split, calculate impurity of split I_{split}.
- Select split with lowest impurity I_{split}.
- Perform split if $I_{split} < I(t)$ (not necessarily always the case).

Impurity of a Node (Classification)
- Gini Index:

$$Gini(t) = 1 - \sum_{c \in C} P(c|t)^2$$

- Entropy:

$$Entropy(t) = - \sum_{c \in C} P(c|t) \log P(c|t)$$

- $P(c|t)$ = relative frequency of class c in node t.

Impurity of a Split (Classification)
- Assume node t is split into k children:
  - n_i — number of records at i-th child.
  - n — number of records at node t.
  - $I(i)$ — impurity of node (e.g., Gini, Entropy).
- Impurity of split:

$$I_{split} = \sum_i \frac{n_i}{n} I(i)$$

- Information Gain:

$$IG = I(t) - I_{split}$$

- Choose split that minimizes I_{split} = split that maximizes IG.
- Required condition: $IG > 0$.

Impurity of a Split (Regression)
- Residual Sum of Squares (RSS):

$$RSS_{split} = \sum_{k=1}^{K} \sum_{i \in R_k} (y_i - \mu_{R_k})^2$$

Decision Trees — Pros & Cons
- **Pros**:
  - Inexpensive to train and test.
  - Easy to interpret (if tree is not too large).
  - Can handle categorical and numerical data.
- **Cons**:
  - Sensitive to small changes in the training data.
  - Greedy approach does not guarantee the optimal tree.
  - Each decision involves only a single feature.
  - Does not take interactions between features into account.

Decision Trees — Overfitting
- Decision Tree algorithm can always split the training data perfectly.
- Solution: Limit size/height of Decision Tree $\Rightarrow$ Pruning.

Pre-Pruning
- **Maximum Depth**
  - Define the maximum depth/height of the tree.
  - Stop splitting if the maximum depth is reached.
- **Minimum Sample Count**
  - Define the minimum number of samples each node must have.
  - Stop splitting if a node has fewer than the minimum number of samples.
  - Example: Minimum sample count = 16.

Bagging (Bootstrap Aggregation)

Bagging — Basic Idea (not limited to Decision Trees)
- Train many models (classifiers/regressors) on different training data.
- Combine predictions of each model for the final prediction.
- Increases accuracy and lowers variance.

Bootstrap Sampling
- Take repeated samples from a single training dataset D.
- Bootstrap sample D_i sampled from D, uniformly and with replacement ($|D_i| = |D|$).
- Train a model over each bootstrap dataset D_i.

Limitations and Consequences
- Strong predictors in D will likely be in most bootstrap samples D_i.
- Most bagged trees will use strong predictors on top.
- Most bagged trees will look very similar.
- Predictions of bagged trees will be highly correlated.
- **Only limited reduction in variance!**

Random Forest

Random Forest = Bootstrap Sampling (Bagging) + Feature Sampling
- Create bootstrap samples D_i like for bagging.
- Feature sampling: For each D_i, consider only a random subset of features of size m.
- Typically, $m \approx \sqrt{d}$.

Effects of Feature Sampling
- Strong predictors in D are often absent in D_i.
- Resulting trees often look very different.
- Predictions of trees are much less correlated.
- **Higher reduction in variance + typically higher accuracy.**

Pros and Cons of Random Forests (Compared to Decision Trees)
- **Pros:**
  - High accuracy—fairly close to state of the art.
  - Sampling and training independent across D_i $\rightarrow$ parallelizable!
  - Not much tuning required.
- **Cons:**
  - Less interpretable.
  - Slower training and prediction.

Boosting

Bagging vs. Boosting
- **Bagging:** Trees are trained independently (can be done in parallel).
- **Boosting:** Trees are trained in sequence; the accuracy of the last tree affects the training of the next tree.
- **Bagging:** All trees have the same amount of say in the final prediction.
- **Boosting:** Trees have different amounts of say in the final prediction (depending on their individual accuracy).

Boosting and Weak Learners
- **Strong Learners:** Perform as best as possible on a given classification or regression task.
- **Weak Learners:** Perform slightly better than guessing.
  - A common weak learner: **Decision Stump** (decision tree of height 1, i.e., only one split).
  - Very simple model $\rightarrow$ very fast training.
- **Boosting:** Combine many weak classifiers into a single strong learner.
  - Basic idea: subsequent models try to improve the errors of previous models.

AdaBoost (Adaptive Boosting)

- Applicable to many classification/regression algorithms to improve performance.
- Very commonly combined with Decision Trees.

Basic Training Algorithm for AdaBoost
- Train a Weak Learner over D_i (e.g., Decision Stump).
- Identify all misclassified samples.
- Calculate the error rate of the learner to quantify its amount of say.
- Sample D_{i+1} such that misclassified samples are more likely to be picked than correctly classified samples.
- Repeat...

Gradient Boosted Trees

- **Gradient Boosting**
  - Mainly applied to regression algorithms to improve performance.
  - Very commonly combined with Decision Trees (for regression).
- **Basic training algorithm**
  - Start with an initial prediction (e.g., mean over all values).
  - Calculate residuals = error between true value and current prediction.
  - Train a Decision Stump to predict residuals.
  - Update predictions based on predicted residuals.
  - Repeat...

Gradient Boosting
- Mainly applied to regression algorithms to improve performance.
- Very commonly combined with Decision Trees (for regression).

Basic Training Algorithm for Gradient Boosting
- Start with an initial prediction (e.g., mean over all values).
- Calculate residuals = error between true value and current prediction.
- Train Decision Stump to predict residuals.
- Update predictions based on predicted residuals.
- Repeat...

Boosting Methods — Pros & Cons (Compared to Decision Trees)
- **Pros**
  - High accuracy — often state of the art.
- **Cons**
  - Less interpretable (arguably even less compared to Random Forests).
  - Slower training and prediction $\rightarrow$ sequential processing $\rightarrow$ not parallelizable.

Algorithm 5 AdaBoost Training Algorithm

```
1: Initialization: Dataset $D$, $|D| = N$, with initial sample weights $w_i = \frac{1}{N}$
2: for $m = 1$ to $M$ do
3:     Generate $D_m$ by sampling from $D$ w.r.t. sampling weights $w$
4:     Train Decision Stump $h_m$ over $D_m$
5:     Apply $h_m$ to all samples in $D$ and identify misclassified samples
6:     Calculate total error: $\epsilon_m = \sum_i^N w_i \cdot \delta(h_m(x_i) \neq y_i)$
7:     Calculate amount of say: $\alpha_m = \frac{1}{2} \ln \frac{1-\epsilon_m}{\epsilon_m}$
8:     Update sample weights:
           $w_i = w_i \cdot \begin{cases} e^{\alpha_m}, & \text{if } x_i \text{ was misclassified} \\ e^{-\alpha_m}, & \text{if } x_i \text{ was correctly classified} \end{cases}$
           $w_i = \frac{w_i}{\sum_i^N w_i}$
9: end for
```

Algorithm 6 Gradient Boosting Training Algorithm

```
1: Initialization: Dataset $D$, $f_0(x_i) = \text{mean}(y)$, $\eta = 0.1$
2: for $m = 1$ to $M$ do
3:     Calculate residuals: $r_{i,m} = y_i - f_{m-1}(x_i)$
4:     Train Decision Stump $h_m$ over $D$ with $r_{i,m}$ as targets
5:     Predicted residuals $h_m(x_i)$ for all training samples
6:     Calculate new predictions: $f_m(x_i) = f_{m-1} + \eta \cdot h_m(x_i)$
7: end for
```

Algorithm 7 Gradient Boosting Training

```
1: Initialization: Dataset $D$, $f_0(x_i) = \text{mean}(y)$, $\eta = 0.1$
2: for $m = 1$ to $M$ do
3:     Calculate residuals: $r_{i,m} = y_i - f_{m-1}(x_i)$
4:     Train Decision Stump $h_m$ over $D$ with $r_{i,m}$ as targets
5:     Predict residuals $\hat{r}_m(x_i)$ for all training samples
6:     Calculate new predictions: $f_m(x_i) = f_{m-1} + \eta \cdot h_m(x_i)$
7: end for
8: Output: $M$ Decision Stumps $h_1, h_2, \ldots, h_M$
```

Lecture 7

Linear Models

Basic setup
- Dataset of n samples: $\{(x_i, y_i)\}_{i=1}^n$
- Input data with d features: $x_i = (x_{i1}, x_{i2}, \ldots, x_{id})$

Assumption
- There exists a linear relationship between x_i and the dependent variable y_i:

$$\hat{y}_i = h_\theta(x_i) = f(\theta_0 x_{i0} + \theta_1 x_{i1} + \theta_2 x_{i2} + \cdots + \theta_d x_{id})$$

$$\theta = \{\theta_0, \theta_1, \theta_2, \ldots, \theta_d\}, \quad \theta_i \in \mathbb{R}$$

Vector Representation
- Introduce constant feature x_{i0}

$$h_\theta(x_i) = f(\theta_0 x_{i0} + \theta_1 x_{i1} + \theta_2 x_{i2} + \cdots + \theta_d x_{id})$$

- Represent x_i with new constant feature:

$$x_i = (1, x_{i1}, x_{i2}, \ldots, x_{id})$$

- Rewrite linear relationship using vectors:

$$h_\theta(x_i) = f(\theta^T x_i)$$

Linear Regression
- **Regression** $\Rightarrow$ Real-valued predictions
- Function f is the identity function $f(x) = x$

$$\hat{y}_i = h_\theta(x_i) = f(\theta^T x_i) = \theta^T x_i$$

$$\hat{y} = X\theta$$

$$\hat{y} = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_n \end{bmatrix}, \quad X = \begin{bmatrix} 1 & x_{11} & x_{12} & \cdots & x_{1d} \\ 1 & x_{21} & x_{22} & \cdots & x_{2d} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & x_{n2} & \cdots & x_{nd} \end{bmatrix}, \quad \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_d \end{bmatrix}$$

Linear Regression — Loss Function
- **Loss function** (also: cost function, error function)
  - Quantifies how good or bad a given set of values for θ is.
  - Measures the difference between predictions $\hat{y}$ and true values y.
- **Mean Squared Error (MSE)**

$$L = \frac{1}{n} \sum_{i=1}^{n} e_i^2 = \frac{1}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2$$

Find Minimum of L Analytically
- Minimum of loss function $L \Rightarrow$ Calculus to the rescue!
  - Partial derivatives w.r.t. to all θ_i are 0:

$$\frac{\partial L}{\partial \theta_0} = 0, \quad \frac{\partial L}{\partial \theta_1} = 0, \quad \ldots, \quad \frac{\partial L}{\partial \theta_d} = 0$$

  - $d + 1$ equations with $d + 1$ unknowns

Rewrite loss function L
- Vector representation makes it easier to handle:

$$L = \frac{1}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2 = \frac{1}{n} \|X\theta - y\|^2$$

- Derive L w.r.t. θ:

$$\frac{\partial L}{\partial \theta} = \frac{2}{n} X^T (X\theta - y)$$

- Set $\frac{\partial L}{\partial \theta} = 0$:

$$\frac{2}{n} X^T (X\theta - y) = \vec{0}$$

Linear Regression — Normal Equation
- Solve for θ:

$$X^T X\theta = X^T y$$

$$(X^T X)^{-1} X^T X\theta = (X^T X)^{-1} X^T y$$

$$\theta = (X^T X)^{-1} X^T y$$

$$\theta = X^\dagger y, \quad \text{with } X^\dagger = (X^T X)^{-1} X^T$$

Pseudo Inverse $X^\dagger$
- $X^\dagger = (X^T X)^{-1} X^T$
- Performance analysis:
  - Most expensive operation: calculating the inverse of $(X^T X)^{-1}$
  - Calculation of inverse depends on number of features d, not on number of data samples n
  - Complexity of calculating inverse of a $d \times d$ matrix: $\mathcal{O}(d^3)$

Analytical Solving Algorithm
- Construct matrix X and vector y from data
- Calculate pseudo inverse $X^\dagger = (X^T X)^{-1} X^T$
- Return $\theta = X^\dagger y$

Gradient Descent
- **Core idea**
  - Start with a random setting of θ
  - Adjust θ iteratively to minimize L

Gradient
- Vector of partial derivatives of a multivariable function (e.g., $\theta_0, \theta_1, \ldots, \theta_d$)
- Partial derivative: slope w.r.t. to a single variable given a current set of values for all $\theta_0, \theta_1, \ldots, \theta_d$
- Points in the direction of the steepest ascent

$$\nabla_\theta L = \frac{\partial L}{\partial \theta} = \begin{bmatrix} \frac{\partial L}{\partial \theta_0} \\ \frac{\partial L}{\partial \theta_1} \\ \frac{\partial L}{\partial \theta_2} \\ \vdots \\ \frac{\partial L}{\partial \theta_d} \end{bmatrix}$$

Gradient Descent Algorithm

- **Important concept: learning rate**
  - Scaling factor for gradient (typical range: 0.01 - 0.0001)

Algorithm 8 Gradient Descent Algorithm

```
1: Input: Data $(X, y)$, loss function $L$, learning rate $\eta$
2: Initialization: Set $\theta$ to random values
3: while not converged do
4:     Compute gradient: $\nabla_\theta L = \frac{\partial L}{\partial \theta}$
5:     Update parameters: $\theta \leftarrow \theta - \eta \cdot \nabla_\theta L$
6: end while
7: Output: Optimal $\theta$
```

In practice: stop loop when θ converges

Gradient Descent — Variations

- **(Basic) Gradient Descent**
  - Calculate gradient and update θ for the whole dataset
- **Stochastic Gradient Descent (SGD)**
  - Calculate gradient and update θ for each data sample
- **Mini-batch Gradient Descent**
  - Calculate gradient and update θ for batches of samples
  - e.g., batch = 64 data samples
  - In practice, often referred to as SGD

Normal Equation vs. Gradient Descent

- **Gradient Descent**
  - Works well even if d is large
  - Works even if $X^T X$ is non-invertible
  - Iterative process; may not find optimal solution in practice
  - Learning rate is a critical hyperparameter
- **Normal Equation**
  - Finds optimal solutions
  - Non-iterative; no need for a learning rate
  - Calculation of $(X^T X)^{-1}$ in $\mathcal{O}(d^3)$

Polynomial Linear Regression

- Linear Regression $\not\Rightarrow$ line / plane / hyperplane
- **Polynomial Linear Regression**
  - Allows capturing nonlinear relationships between X and y
  - Polynomial regression model for 1 input feature

$$\hat{y}_i = \theta_0 1 + \theta_1 x_i + \theta_2 x_i^2 + \cdots + \theta_p x_i^p$$

Matrix representation (again, 1 input feature!)

$$X^{(1)} = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_n \end{bmatrix} \quad X^{(2)} = \begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ \vdots & \vdots & \vdots \\ 1 & x_n & x_n^2 \end{bmatrix} \quad X^{(3)} = \begin{bmatrix} 1 & x_1 & x_1^2 & x_1^3 \\ 1 & x_2 & x_2^2 & x_2^3 \\ \vdots & \vdots & \vdots & \vdots \\ 1 & x_n & x_n^2 & x_n^3 \end{bmatrix}$$

Polynomial Linear Regression — Overfitting

- **Increasing degree of polynomial** p
  - More capacity to capture nonlinear relationships
  - Much higher sensitivity to noise and outliers
- **Countermeasure: Regularization**
  - Extend loss function to "punish"large values of θ

$$L = \frac{1}{n}\|X\theta - y\|^2 + \lambda \frac{1}{n}\|\theta\|_2^2$$

$$\|\theta\|_2^2 = \sum_{i=1}^{d} \theta_i^2$$

Note: excludes θ_0!

Polynomial Linear Regression — Minimizing Loss L

- **Normal Equation**

$$\theta = (X^T X + \lambda \begin{bmatrix} 0 & & & \\ & 1 & & \\ & & \ddots & \\ & & & 1 \end{bmatrix})^{-1} X^T y$$

- **Gradient Descent**

$$\nabla_\theta L = \frac{2}{n} X^T (X\theta - y) + \lambda \frac{2}{n}\theta$$

Polynomial Linear Regression — More than 1 Feature

- Number of terms in multivariate polynomial given p, d

$$\theta_i, \quad 0 \leq i \leq M, \quad \text{with } M = \binom{p+d}{p}$$

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

- **Practical considerations**
  - Limited to small number of features and small polynomial degrees
  - In principle, terms can be dropped (e.g., all interaction terms)

Linear Regression — Interpretation of θ_i

- Change of value of feature i by 1 unit $\Rightarrow$ change of output value by θ_i
- Assumption: all other feature values remain the same

Data Normalization — Yes or No? (*standardization, min-max scaling*)

Data normalization does not affect model performance

(assuming basic Linear Regression without regularization)

In favor of "No"

- Preserves unit of feature $i \Rightarrow$ direct interpretation of θ_i
- Better for comparing θ_i for the same features across different datasets

In favor of "Yes"

- When using regularization
- When using Polynomial Linear Regression
- Better for comparing θ_i within a model

$$(\theta_i > \theta_j \Rightarrow \text{feature } i \text{ more important than feature } j)$$

Logistic Regression

- Logistic Regression $\Rightarrow$ Real-valued predictions interpreted as probability
- Function f is the standard **Logistic Function** (Sigmoid function)

$$f(x) = \frac{L}{1 + e^{-k(x - x_0)}}$$

For $L = 1, k = 1, x_0 = 0$:

$$f(x) = \frac{1}{1 + e^{-x}}$$

Logistic Regression: Probabilistic Interpretation

- $\hat{y}$ interpreted as a probability

$$\hat{y} = h_\theta(x) = f(\theta^T x) = \frac{1}{1 + e^{-\theta^T x}}, \quad \hat{y} \in [0, 1]$$

- $\hat{y} = h_\theta(x)$ is the estimated probability that $y_i = 1$ given x and θ

$$\hat{y} = P(y = 1|x, \theta)$$

Given only discrete 2 outcomes:

$$P(y = 1|x, \theta) + P(y = 0|x, \theta) = 1$$

$$\hat{y} = 1 - P(y = 0|x, \theta)$$

$$\hat{y} = P(y = 1|x, \theta) = 1 - P(y = 0|x, \theta)$$

- $P(y|x)$ is a Bernoulli distribution

$$P(y|x) = \begin{cases} \hat{y}, & y = 1 \\ 1 - \hat{y}, & y = 0 \end{cases}$$

$$P(y|x) = \hat{y}^y (1 - \hat{y})^{1-y}$$

Logistic Regression: Loss Function

Goal: Maximize probability of true y label given training sample x

- Find θ that maximizes

$$P(y|x) = \hat{y}^y (1 - \hat{y})^{1-y}$$

$$\log P(y|x) = \log \left[\hat{y}^y (1 - \hat{y})^{1-y}\right] = y \log \hat{y} + (1 - y)\log(1 - \hat{y})$$

- Find θ that minimizes

$$L = -P(y|x) = -[y \log \hat{y} + (1 - y)\log(1 - \hat{y})]$$

Cross-Entropy Loss
Logistic Regression: Loss Function
- Loss for all training samples

$$L = -\frac{1}{n}\sum_{i=1}^{n}[y_i \log \hat{y}_i + (1-y_i)\log(1-\hat{y}_i)]$$

$$= -\frac{1}{n}\sum_{i=1}^{n}[y_i \log h_\theta(x_i) + (1-y_i)\log(1-h_\theta(x_i))]$$

$$= -\frac{1}{n}\sum_{i=1}^{n}\left[y_i \log \frac{1}{1+e^{-\theta^T x_i}} + (1-y_i)\log\left(1-\frac{1}{1+e^{-\theta^T x_i}}\right)\right]$$

Gradient of the Loss Function
- After lots of tedious math...

$$\frac{\partial L}{\partial \theta_j} = \frac{1}{n}\sum_{i=1}^{n}[h_\theta(x_i)-y_i]x_{i,j}$$

$$\nabla_\theta L = \frac{1}{n}X^T(h_\theta(X)-y)$$

- Problem:

$$\frac{1}{n}X^T(h_\theta(X)-y) \neq 0$$

has no closed-form solution for θ
⇒ **Gradient Descent!**
Polynomial Logistic Regression
- Analogous to Polynomial Linear Regression
  - Allows to capture nonlinear relationships between X and y
  - Polynomial Logistic Regression model for 1 input feature

$$\hat{y}_i = \frac{1}{1+e^{-\theta^T x_i}}, \quad \text{with } \theta^T x_i = \theta_0 + \theta_1 x_i + \theta_2 x_i^2 + \ldots + \theta_p x_i^p$$

Identical practical considerations
- Solve using Gradient Descent as usual (optionally with regularization to avoid overfitting)
- Limited to small number of features and small polynomial degrees

$$\nabla_\theta L = \frac{1}{n}X^T(h_\theta(X)-y) + \frac{\lambda}{n}\theta$$

Lecture 8

Recommender System

Motivation
- In the online world: information and item overload
  - Too many items: products, songs, movies, news articles, restaurants, etc.
  - More choices require better filters ⇒ recommendation engines

Manual Recommendations — Pros & Cons
- **Pros**
  - Semantically rich (ratings, plain text, images, videos, etc.)
  - Explainability / Interpretability
- **Cons**
  - Manual effort — What is the incentive for writing a review?
  - Lack of personalization

Simple Aggregations
- Rank items based on aggregated
- **Pros**
  - Relatively easy to compute (typically weighted aggregated based on different factors)
  - Typically good/safe recommendations (particularly for new/unknown users)
- **Cons**
  - Requires sufficient number of ratings per item
  - High risk of popularity bias; lack of diversity ("rich get richer"effects, "few get richer"effects)
  - Lack of personalization

Personalized Recommendations
- Users have different preferences that define the relevance of items
  - Preferences = interests, likings, needs, wants, desires, etc.
  - Relevant items = items that match users' preferences best

Basic Setup
- Set of users $U = \{u_1, u_2, ..., u_n\}$
- Set of items $V = \{v_1, v_2, ..., v_m\}$
- Rating matrix R with $|U|$ rows and $|V|$ columns
- Matrix element $R_{u,v}$: u's rating of v (e.g., 1-5 stars, binary 0/1)

Personalized Recommendations — Core Tasks
- Collect ratings R_{uv}
- Infer missing values $\hat{R}_{uv}$
  - In practice, mainly interested in high values
  - Algorithmic component of recommender systems
  - Wide range of existing approaches
- Evaluation: How good are the recommendations?
  - Compare $\hat{R}_{uv}$ with true R_{uv} from test set

Collecting Ratings
- **Explicit**
  - Ask/invite/encourage users to rate items
  - Pay users to rate items (e.g., crowdsourcing)
- **Implicit — derive ratings from users' behavior, e.g.:**
  - Product bought
  - Video watched
  - Article read
  - Link clicked
  - ...
- **Key challenge**: Rating matrix R is in practice very sparse!

Evaluation
- Split R into training and test set
- Performance metrics:
  - Root Mean Squared Error (for numerical ratings)
  - Precision, Recall, F1 score, etc. (TP, TN, FP, FN for binary ratings or binary recommendation after converting numerical ratings)
  - Precision@k, Recall@k (precision and recall w.r.t. the top-k highest predicted ratings)
  - Compare rankings induced by $\hat{R}_{uv}$ and R_{uv} with $(u,v) \in S$

Recommendations Using Association Rules
- **User preferences & likings**
  - Items: movies, songs, books, etc.
  - Transaction: viewing/listening/reading history
- **Interesting rules (movies)**
  - Viewer who watched movies $\{a, b\}$ also watched movies $\{x, y\}$
  - Example: $\{Jaws\} \rightarrow \{It\}$
- **Limitations**
  - Basic AR algorithm ignores ratings
  - Popularity bias: user with very unique tastes likely to get subpar recommendations

Recommendations Using Clustering
- **Approach**
  - Cluster movies based on "useful"features (genre, director, writer, length, ...)
  - Recommend movies from clusters with movies a user has rated highly
- **Limitations**
  - Find good features in practice is very difficult
  - Unsystematic: no well-defined process to pick recommendations

Recommendations Using Regression (or Classification)
- **Example approach: Linear Regression**
  - Independent variable: movie features
  - Dependent variable: user rating
- Build a linear regression model for each user
- **Limitations**
  - Requires good features for each item
  - Cold-start problem: requires a lot of user ratings to build a good model

Content-Based Recommender System
Intuition
- Recommend item v to user u that are similar to v and u has rated highly.
- Examples:
  - Movies of the same genre.
  - Songs from the same artist.
  - Articles about the same topic.
  - Products with similar features.

Basic Requirement: Item Profiles = Feature Vector for Each Item
- Movie: genre, director, writer, cast, length, year, ...
- Product: type, brand, price, weight, color, ...
- Article: set of (important) words / tf-idf vector / ...

Simple Approach — Pairwise Item Similarity
- Pairwise item similarity $sim(x, y)$
  - x, y — feature vectors of movies.
  - Common metric: cosine similarity:

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

Limitation: Requires Reference Item
- Movie(s) the user was most recently watching.
- Movie(s) the user has rated the highest.
- Movie(s) the user is currently browsing.

User-Item Similarities
Needed: User Profiles = Feature Vector for Each User
- Requirement: same shape as item profiles to calculate similarities.
- Approach: user profile = "some aggregate"over item profiles rated by the user.

User-Item Similarities — Binary Utility Matrix
- $R_{uv} \in \{0, 1\}$ — for example, $R_{uv} = 1$ if:
  - User u bought movie v.
  - User u watched movie v.
- Implicit rating that u likes v (no explicit ratings available here; but also no implicit dislikes!).

Simple Average
- Computes average across binary ratings:

$$\text{User profile} = \frac{1}{|M|}\sum_{v \in M} \text{feature vector of } v$$

where M is the set of movies the user has rated.

User-Item Similarities — Real-Valued Utility Matrix
- $R_{uv} \in \mathbb{R}$ — for example, $R_{uv} \in \{1.0, 1.5, 2.0, ..., 5.0\}$ star rating.
- Explicit rating of user u for movie v.
- Important: semantic interpretation — ratings express both likes and dislikes.
- Use rating as weights for features for a weighted aggregation:

$$w_f = \frac{\sum_{v \in M} R_{uv} \cdot f_v}{\sum_{v \in M} R_{uv}}$$

where w_f is the weight of feature f, and f_v is the feature value for item v.

Intuition
- The user likes romantic and animated movies.
- The user dislikes fantasy and adventure movies.

Step 1: Normalize Ratings
- Subtract average user rating from each movie rating:

$$R'_{uv} = R_{uv} - \frac{1}{|M|} \sum_{v \in M} R_{uv}$$

- Converts ratings into positive (liked) and negative (disliked) scale.
- Distinguishes "generous" users (mostly rate highly and 3.0 is a low rating) from more "grumpy" users (mostly rate low and 3.0 is a high rating).

Step 2: Calculate Weighted Features for User Profile
- The weights are the normalized weights:

$$w_f = \frac{\sum_{v \in M} R'_{uv} \cdot f_v}{|M|}$$

User-Item Similarities
Pairwise Item Similarity $sim(u, v)$
- u — user profile; v — item profile.
- Suitable metric: cosine similarity:

$$sim(u, v) = \frac{u \cdot v}{\|u\| \|v\|}$$

Recommend items v_i to user u with max. similarities $sim(u, v_i)$
Practical Considerations
- Top k most similar items always the same $\Rightarrow$ add some randomization for diversity.
- Top k most similar items might include items the user has already rated $\Rightarrow$ remove those items.
- More sophisticated ways to aggregate item profiles to user profiles conceivable (e.g., ignore underrepresented features).

Content-Based Recommender System — Pros & Cons
Pros
- Recommendations for user u do not depend on other users.
- Recommendations can also include new or unpopular items.
- Good explainability (features that had most effect on the high similarity).

Cons
- Cold-start problem: How to build a profile for new users?
  - Naive approach: recommend generally popular items to new users.
- Finding good features (and values!) for items is a non-trivial task.
- Overspecialization: By default, no recommendations outside a user's profile.
  - In practice: add some randomization into the recommendation process.

Collaborative Filtering
- **Idea: Utilize the opinions of others**
  - Recommend items that other users with similar tastes/preferences/needs have liked
  - Does not require item or user-specific features
- **Two perspectives**
  - User-based — two users are similar if they rated the same items similarly
  - Item-based — two items are similar if they are equally rated by users

User-Based Collaborative Filtering — Calculating Similarities
- **Represent all users by their rating vectors**
  - Rows of rating matrix:

$$r_A = (2, 4, 5, 0, 1)^T$$

$$r_B = (1, 0, 4, 0, 2)^T$$

- **Using Cosine Similarity**
  - Example calculations:

$$\text{sim}(r_A, r_B) = 0.77$$

- **Problem with this approach**
  - Missing values (0) are treated as negative
  - All ratings are positive values
  - No explicit notion of dissimilarity (only less or more similar)

User-Based Collaborative Filtering — Normalizing Ratings
- **Idea: Normalize rating vectors**
  - Mean-centering — subtract row mean from each rating vector
  - Missing values (0) now represent the average rating
  - Bad ratings (i.e., below averaged) now represented by negative values
- **Cosine similarity between mean-centered vectors**
  - Uses the Pearson Correlation Coefficient
  - Example:

$$\text{sim}(r_A, r_B) = 0.78$$

$$\text{sim}(r_D, r_B) = -0.65$$

User-Based Collaborative Filtering — Predicting Ratings
- Estimated rating $\hat{R}_{uv}$ is the weighted average of ratings from similar users

$$\hat{R}_{uv} = \frac{\sum_{w \in N} \text{sim}(u, w) \cdot R_{wv}}{\sum_{w \in N} \text{sim}(u, w)}$$

- N is the set of k users most similar to u who have already rated item v

Item-Based Collaborative Filtering
- Analog to user-based approach
  - Find the most similar items
  - Two items are similar if their ratings across all users are similar
- **Predicting Ratings**

$$\hat{R}_{uv} = \frac{\sum_{i \in M} \text{sim}(i, v) \cdot R_{ui}}{\sum_{i \in M} \text{sim}(i, v)}$$

- M is the set of k items most similar to v that have already been rated by u

Collaborative Filtering — User-Based vs. Item-Based
- In theory, user-based and item-based are dual approaches
- In practice, item-based typically outperforms user-based
  - Items are "simpler" than users
  - Items can be more easily described
  - Users can have very varied tastes
- Item-item similarity is typically more meaningful.

Model-Based Collaborative Filtering
- **Latent Factor Models**
  - Latent representation: k-dimensional vector for each user u and item v
  - Learn latent representations from the data
  - Estimate unknown ratings:

$$\hat{R}_{uv} = w_u^T h_v$$

- **Matrix Factorization Approach**
  - Put all user vectors into a matrix W
  - Put all item vectors into a matrix H
  - Find W, H such that:

$$R \approx WH$$

Finding Matrices W, H
- **Minimize loss function**

$$L = \sum_{R_{uv}>0} e_{uv} = \sum_{R_{uv}>0} (R_{uv} - \hat{R}_{uv})^2 = \sum_{R_{uv}>0} (R_{uv} - w_u^T h_v)^2$$

- **With regularization**

$$L = \sum_{R_{uv}} (R_{uv} - w_u^T h_v)^2 + \lambda(\|w_u\|^2 + \|h_v\|^2)$$

- **Using Gradient Descent**
  - Compute gradients:

$$\frac{\partial e_{uv}}{\partial w_u} = -2(R_{uv} - w_u^T h_v) h_v + 2\lambda w_u$$

$$\frac{\partial e_{uv}}{\partial h_v} = -2(R_{uv} - w_u^T h_v) w_u + 2\lambda h_v$$

  - Update rules:

$$w_u \leftarrow w_u - \eta \frac{\partial e_{uv}}{\partial w_u}$$

$$h_v \leftarrow h_v - \eta \frac{\partial e_{uv}}{\partial h_v}$$

Collaborative Filtering — Pros & Cons
- **Pros**
  - No need to find and create good features (e.g., genres for movies)
  - Intuitive approach
- **Cons**
  - Similarity calculations rely on sufficient number of ratings
  - Cold-start problem in case of new users or items
  - Popularity bias: users with unique tastes likely receive subpar recommendations
  - Naive implementation is expensive: finding k most similar users/items has complexity $O(|R|)$

Lecture 9

Graph: Formalism for representing *relationships* between items

A graph is a tuple $G = (V, E)$

- Set of vertices (or nodes) V $\quad V = \{v_1, v_2, \ldots, v_n\}$
- Set of edges E $\quad E = \{e_1, e_2, \ldots, e_m\}$, where an edge is a pair of vertices: $e_i = (v_j, v_k)$

Example:

$$V = \{A, B, C, D\} \quad E = \{(A, B), (A, C), (C, D), (B, A), (C, B)\}$$

Types of Graphs

- **Directed vs Undirected:**
  - *Undirected:* Edges have no direction (e.g., Facebook friendships)
  - *Directed:* Edges have a direction (e.g., Twitter followers)
- **Weighted vs Unweighted:**
  - *Unweighted:* All edges are equal (e.g., MRT connectivity)
  - *Weighted:* Edges have weights (e.g., travel time, number of co-authored papers)
- **Cyclic vs Acyclic:**
  - *Cyclic:* Graph has at least one cycle
  - *Acyclic:* No cycles exist
- **Simple vs Multigraph:**
  - *Simple Graph:* At most one edge between a pair of nodes
  - *Multigraph:* Multiple edges allowed between same pair of nodes
- **Sparse vs Dense:**
  - *Sparse:* Relatively few edges
  - *Dense:* Many edges, close to complete graph
- **Connected Graphs:**
  - *Strongly Connected:* Directed graph where a path exists between all pairs of nodes
  - *Weakly Connected:* Underlying undirected graph is connected
  - *Disconnected:* Not all nodes are connected

Adjacency Matrix: A matrix A used to represent a graph. Entry A_{ij} is nonzero if there is an edge from node i to node j.

Example (Unweighted Undirected):

$$\text{Graph:} \quad V = \{A, B, C, D\} A = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Example (Weighted Directed):

$$A = \begin{bmatrix} 0 & 2 & 3 & 0 \\ 3 & 0 & 0 & 0 \\ 0 & 5 & 0 & 10 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Community Detection No formal definition of a community. From "Networks: An Introduction"(Mark Newman): *"Loosely stated, [community detection] is the problem of finding the natural divisions of a network into groups of vertices such that there are many edges within groups and few edges between groups. What exactly we mean by "many" or "few," however, is debatable."*

Wide range of application

- Identifying groups in social networks
- Recommendation systems
- Market segmentation
- Outlier/anomaly detection

Community Detection — Modularity

Modularity $Q \in [-\frac{1}{2}, 1]$ of an undirected graph G with adjacency matrix A. Measures the relative density of edges inside communities with respect to edges outside communities. Optimizing modularity is NP-hard → practical algorithms based on heuristics.

$$Q = \frac{1}{2m} \sum_{vw} \left[A[v, w] - \frac{k_v k_w}{2m} \right] \delta(c_v, c_w)$$

where:

- $A[v, w]$: weight of edge between nodes v and w
- k_i: sum of weights of edges attached to node i
- c_i: community of node i
- $\delta(c_v, c_w) = \begin{cases} 1 & c_v = c_w \\ 0 & \text{otherwise} \end{cases}$

Community Detection — Louvain Algorithm

Algorithm 9 Louvain Algorithm

```
1:  Initialization: Each node is a community
2:  repeat
3:      Phase 1: Modularity Optimization
4:      for each node $v$ do
5:          Check if moving $v$ to an adjacent community improves modularity
6:          Move $v$ to community that maximizes modularity
7:      end for
8:      Phase 2: Graph Aggregation
9:      Represent each community as a new node
10:     Update weights between new nodes
11: until no further change
```

Louvain Algorithm — Remarks

Heuristic

- Optimizes modularity locally on all nodes
- No guarantees for optimal modularity globally (in practice often superior to other methods)

Performance optimization

- Phase 1 requires computing change in modularity ΔQ
- ΔQ can be computed based on local changes in community assignments (no need to recompute modularity after each change)

Community Detection — Girvan-Newman Algorithm

Divisive hierarchical approach

- Start with whole graph representing a community
- Iteratively remove edges until the community is split into two sub-communities (can recurse)

Criteria for removing edges: Edge Betweenness Centrality

$$c_B(e) = \sum_{v, w \in V} \frac{\sigma(v, w|e)}{\sigma(v, w)}$$

where:

- $\sigma(v, w)$: number of shortest paths from v to w
- $\sigma(v, w|e)$: number of those paths that pass through edge e

Algorithm 10 Girvan-Newman Algorithm

```
1: repeat
2:     Calculate $c_B(e)$ for all $e \in E$
3:     Remove edge with max $c_B(e)$
4: until graph is split into two components
```

Recursive step

- Apply algorithm to each new component
- Stop if a component contains only a single node (or based on user-specified stopping)

Girvan-Newman Algorithm — Remarks

Complexity Analysis

- Core concept: Edge Betweenness Centrality
- Requires solving the All-Pairs Shortest Path (APSP) problem
- Time complexity depends on graph type (directed/undirected, cyclic/acyclic, weighted/unweighted, etc.)

Min-Cut Problem

- Given a graph G, cut G into 2 components such that the number of edges between both components is minimal.
- Example: $|\text{Min-Cut}| = 2$ in the illustrated graph.
- Fundamental problem in graph theory ⇒ many existing algorithms (focus depends on directed vs. undirected, etc.).

Karger's Algorithm

- **Randomized** method to find Min-Cut.
- Applicable to undirected graphs with positive weights (includes unweighted graphs).

Algorithm 11 Karger's Min-Cut Algorithm

```
while $|V| > 2$ do
    Randomly pick a remaining edge $e = (v, u)$
    Merge/contract $v$ and $u$ into a new node
      - Update edges to neighbors of $v$ and $u$
      - Remove self-loops
end while
Return edges between the final 2 nodes as Min-Cut
```

Intuition: Edges that are in the Min-Cut have a lower probability of being picked. *Runtime:* $O(|V|^2)$ (basic), but further optimizations exist.

Analysis

- What is the probability that the algorithm finds the *correct* Min-Cut?
- For an undirected graph $G = (V, E)$ with $n = |V|$ and $m = |E|$:

$$\text{Average degree} = \frac{1}{n} \sum_{v \in V} \text{degree}(v) = \frac{2m}{n}$$

$$|\text{Min-Cut}| \le \frac{2m}{n} \Rightarrow P(\text{edge is in Min-Cut}) \le \frac{2}{n}$$

- Let $P(\text{success}) = P(\text{final cut is Min-Cut})$:

$$P(\text{success}) \ge \left(1 - \frac{2}{n}\right)\left(1 - \frac{2}{n-1}\right) \ldots \left(1 - \frac{2}{3}\right) = \frac{2}{n(n-1)} = \left(\binom{n}{2}\right)^{-1}$$

Repeated Runs and Total Runtime

- If run k times and smallest cut chosen, what is $P(\text{failure})$?

$$P(\text{failure}) = \left[1 - \left(\binom{n}{2}\right)^{-1}\right]^k$$

$$\text{With } k = \binom{n}{2} \ln n \Rightarrow P(\text{failure}) \leq \left(\frac{1}{e}\right)^{\ln n} = \frac{1}{n}$$

- Total runtime $O(n^2 \log n) \times O(n^2) = O(n^4 \log n)$

Remarks

- In general, a graph has multiple possible Min-Cuts.
- Choice is application-specific:
  - Favor Min-Cuts where the two components are of similar size.
  - Ignore Min-Cuts where a component size is below a threshold.

Centrality Measures: Quantify the **importance** of a node given its topological position in a graph. Centrality is usually assigned to nodes but can also be extended to edges (e.g., Edge Betweenness Centrality). Different measures capture different "flavours"of importance.

What makes a node important? That depends on the specific notion of importance the centrality measure encodes.

Applications:

- Identify influential social network users
- Identify superspreaders of diseases
- Identify key nodes in infrastructure networks

Degree Centrality: A local measure; counts how many edges are directly incident to a node.

Undirected Graph:

$$c_d(v_i) = \sum_{v_j \in V} A[i,j]$$

Directed Graph:

$$c_{d\_in}(v_i) = \sum_{v_j \in V} A[j,i], \quad c_{d\_out}(v_i) = \sum_{v_j \in V} A[i,j]$$

Note: For unweighted graphs, $A[i,j] = 1$ for an existing edge. Thus, the sum equals the degree.

Pros:

- Simple and efficient to compute
- Often sufficient for basic applications

Cons:

- Only considers immediate neighbors
- All edges treated equally
- Vulnerable to manipulation (e.g., spam links or fake accounts)

Examples of manipulation:

- Fake websites linking to boost ranking
- Fake social media followers
- Fake reviews on e-commerce platforms

Eigenvector Centrality: A recursive measure—nodes are important if they connect to other important nodes. Computed using the principal eigenvector of the adjacency matrix A:

$$c_{ev}(v_i) = \frac{1}{\lambda} \sum_{v_j \in V} A[i,j] \cdot c_{ev}(v_j), \quad \lambda c_{ev} = A c_{ev}$$

Power Iteration Algorithm (to compute largest eigenvector):

Algorithm 12 Power Iteration Method

```
1: Input: Matrix $M$, error threshold $\epsilon$, max iterations $T$
2: Initialize: $t = 0$, $x_0 = [1/|V|, 1/|V|, \dots]$
3: repeat
4:     $t \leftarrow t + 1$
5:     $x_t \leftarrow M x_{t-1}$
6:     $x_t \leftarrow x_t / \|x_t\|$                    Normalize
7:     $\delta \leftarrow \|x_t - x_{t-1}\|$              Compute change
8: until $\delta < \epsilon$ or $t > T$
9: Return $x_t$
```

PageRank: Eigenvector-like measure adapted for directed graphs (e.g., web graphs). Uses a *Random Surfer* model:

$$c_{pr}(v_i) = \alpha M c_{pr}(v_i) + (1-\alpha)E, \quad E = \left[\frac{1}{|V|}, \dots\right]^T$$

Where M is the transition matrix of the graph. M is column-stochastic so $\lambda = 1$.

Remarks on Eigenvector-Based Measures:

- Recursive and intuitive definition
- Extended models: HITS, SALSA, Katz, personalized PageRank
- Costlier to compute but highly parallelizable

Closeness Centrality: A node is central if it is "close" to all others (short average distance).

$$c_{cl}(v) = \frac{N}{\sum_{w \in V} d(v,w)}$$

Where $d(v,w)$ is the shortest path length. Works on directed/undirected graphs.

Betweenness Centrality: Measures how often a node appears on shortest paths between other pairs:

$$c_b(v) = \sum_{s,t \in V; s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

Where $\sigma_{st}(v)$ is the number of shortest paths from s to t that pass through v.

Remarks on Closeness & Betweenness:

- Both rely on **shortest path distances**
- Require solving the **All-Pairs Shortest Path (APSP)** problem
- Complexities vary by graph type (e.g., cyclic vs acyclic, with or without negative edges)

Summary of Popular Centrality Measures:

- **Local:** Degree, InDegree, OutDegree
- **Eigenvector-based:** Eigenvector Centrality, PageRank
- **Distance-based:** Closeness, Betweenness

Lecture 10

Sampling — Basic Definition

- Process of selecting members of a population of interest (e.g., HDB residents)
- Consideration of whole population typically impractical (e.g., too costly to survey all HDB residents)
- Goal: statistical analysis of population (e.g., average happiness, most common complaints)

Sampling Data Streams

- Population = stream of incoming data items
- Time and/or resource constraints generally make it impossible to consider all data items
- Relevant patterns based on statistical analysis

Core Challenge: How to get a representative sample?

Problems with Naive Approach — Toy Example Setup

- Stream of search queries — items are tuples (user, query, time)
- Goal: Fraction of queries issued more than once (twice) in last 24h
- Restriction: only 10% of all tuples should be stored

Naive Approach

- Store latest tuple with probability 1/10
- Let s = number of times a user issued a query once
- Let d = number of times a user issued a query twice

Correct Estimate:

$$\frac{d}{s+d}$$

Estimate based on Naive Sample:

$$\frac{d}{10s + 19d}$$

Problems with Naive Approach — Explanation

The Issue:

- Of d queries issued twice:
  - $\frac{d}{100}$ will be both in the sample
  - $\frac{18d}{100}$ will be only once in the sample

$$\left(\frac{1}{10}\right)\left(\frac{1}{10}\right) d = \frac{1}{100} d$$

$$2\left(\frac{1}{10}\right)\left(\frac{9}{10}\right) d = \frac{18}{100} d$$

Fractions of Queries Issued Twice in the Sample:

$$\frac{\frac{1}{100} d}{\frac{1}{100} d + \frac{18}{100} d + \frac{1}{10} s} = \frac{d}{10s + 19d}$$

Sample with a Given Probability

General Approach

- Given items/tuples with n components (e.g., user, query, time)
- Select subset of components as key for sampling
- Key choice depends on goal of analysis

Question: How to obtain a sample consisting of any fraction a/b of keys?

Common Implementation via Hash Function

- Define hash function h that maps $h(key)$ into b buckets $1 \dots b$
- For each new tuple, if $h(key) \leq a$ (with $a \leq b$), add to sample

Sample with a Given Size Limit — Reservoir Sampling

Goal: Maintain a uniform random sample of fixed size B

- Uniform random = each item has same probability to be sampled
- Allows approximation of statistics like mean, variance, median, etc.

Basic Algorithm
- Input stream of items $\{a_1, a_2, \dots\}$
- Maximum reservoir size B
- Each item sampled with probability B/t at any time t

Algorithm Steps:
- Add a_t to reservoir if $t \leq B$
- When receiving a_t with $t > B$:
  - With probability B/t, replace a random item in reservoir with a_t

Bloom Filters
- **Basic Idea:**
  - Create bit array B with $1..n$ bits and $B[i] = 0$
  - Choose m independent hash functions $h_i(key) \in [1, n]$
  - For each key $s \in S$, set $B[h_i(s)] = 1$, for all $1 \leq i \leq m$
- **Example:**
  - 3 hash functions: h_1, h_2, h_3
- **Filter step for key k:**
  - Accept if $B[h_i(k)] = 1$ for all $1 \leq i \leq m$, discard otherwise

False Positive Rate — Analysis (Bloom Filter)
- Probability of $B[h_i(s)] = 1$ with $s \notin S$:

$$1 - e^{-\frac{|S|}{|B|}}$$

- Probability for all m hash functions:

$$\left(1 - e^{-\frac{m|S|}{|B|}}\right)^m$$

False Positive Rate — Visualization (Bloom Filter)
- Effect of number of hash functions m on false positive rate
- **Global Optimum:**
  - Larger m implies more conditions to be satisfied
  - But more 1s in the bit array increases collisions
- **Optimal value for m:**

$$m_{best} = \frac{|B|}{|S|} \ln 2$$

- Example: $m_{best} = 5.5 \approx 6$, giving false positive rate around 2.2%

Bloom Filter — Discussion
- **Memory and Space Consumption**
  - Time complexity: $O(m)$
  - Space complexity: $O(|B|)$
- **Limitation: No Support for Removing Keys**
  - E.g., cannot remove a whitelisted email address
  - Only workaround: Rebuild the lookup table from scratch

Extension — Counting Bloom Filters
- Replace bits by counters (typically 3–4 bits per counter)
- Increase lookup table size (3–4 times)

Operations:
- Insert: $B[h_i(s)] += 1$, for all $1 \leq i \leq m$
- Delete: $B[h_i(s)] -= 1$, for all $1 \leq i \leq m$
- Lookup: Accept if $B[h_i(k)] > 0$ for all $1 \leq i \leq m$

False Positive Rate:
- Same as standard Bloom filter:

$$\left(1 - e^{-\frac{m|S|}{|B|}}\right)^m$$

Counting Unique/Distinct Elements
- **Applications:**
  - Number of unique Facebook visitors (identified by user ID)
  - Number of unique website visitors (identified by IP address)
  - Number of unique words in a large text document
- **Straightforward Solution:**
  - Maintain set S of items seen so far
  - Number of distinct elements $\to |S|$

Problem: What if S can grow very large?

Flajolet-Martin Algorithm
- **Approximation Approach:**
  - Estimate distinct count in an unbiased way
  - Accept errors but minimize their probability
- **Basic Algorithm:**
  1. Choose hash function that maps each of the n elements to at least $\log_2 n$ bits
  2. For each element s in stream:
     - Calculate $h(s)$: bit string of length $\log_2 n$
     - Let $r(s)$ = number of trailing 0s in $h(s)$
     - Keep track of largest $r(s)$, call it R
  3. Return estimate as 2^R

Flajolet-Martin Algorithm — Intuition & Proof Sketch
- **Basic Intuition:**
  - More distinct elements $\to$ more different hash values
  - Unusual hash value = rare bit pattern (e.g., many trailing zeros)

Probability that $h(a)$ ends in at least k trailing 0s:

$$\left(\frac{1}{2}\right)^k$$

Probability that $h(a)$ ends in less than k trailing 0s:

$$1 - \frac{1}{2^k}$$

Given m distinct elements:
Probability that all $h(a)$ end in less than k trailing 0s:

$$\left(1 - \frac{1}{2^k}\right)^m$$

Probability that at least one $h(a)$ has at least k trailing 0s:

$$1 - \left(1 - \frac{1}{2^k}\right)^m$$

Flajolet-Martin Algorithm — Proof Sketch (Continued)

$$1 - \left(1 - \frac{1}{2^k}\right)^m \approx 1 - e^{-m/2^k}$$

Case 1: $2^k \ll m$

$$1 - e^{-m/2^k} \approx 1$$

Case 2: $2^k \gg m$

$$1 - e^{-m/2^k} \approx m/2^k \approx 0$$

Flajolet-Martin Algorithm — Proof Sketch (Interpretation)
- **Case 1:** $2^k \ll m$
  - Probability to get an $h(a)$ with enough trailing 0s is high
- **Case 2:** $2^k \gg m$
  - Probability to get an $h(a)$ with too many trailing 0s is low
- **Conclusion:** R is typically in the right ballpark

Flajolet-Martin Algorithm — Problems & Extensions
- **Problem:**
  - Estimate is always a power of 2
  - If R is off by 1, the estimate doubles or halves
- **Practical Solution:**
  - Use multiple hash functions $h_i(a)$
  - $p \times q$ hash functions $\to$ $p \times q$ estimates
  - Group estimates into p groups of q values
  - Calculate median of each group $\to$ p medians
  - Calculate mean over all p medians