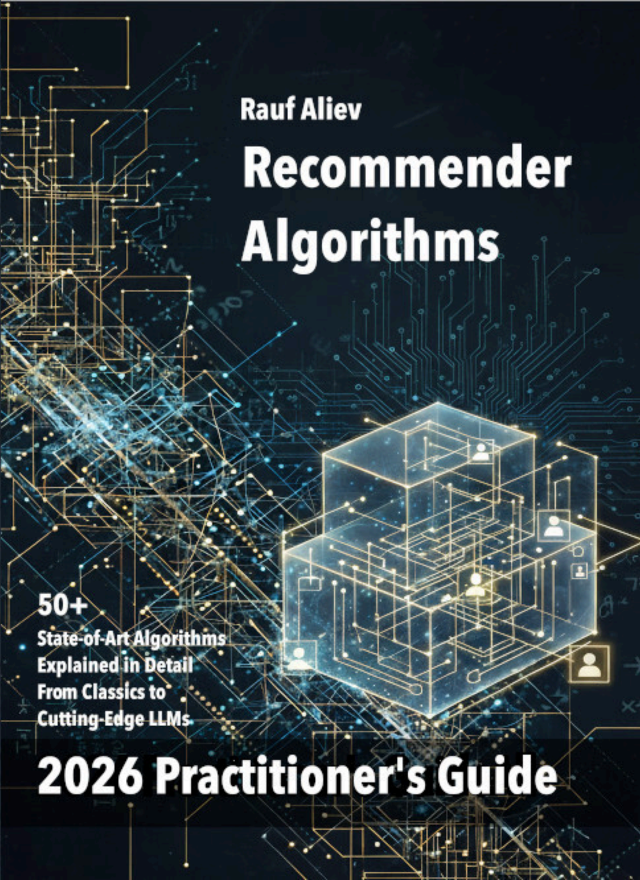
Rauf Aliev

Recommender Algorithms



50+
State-of-Art Algorithms
Explained in Detail
From Classics to
Cutting-Edge LLMs

2026 Practitioner's Guide

Recommender Algorithms

2026 Practitioner's Guide

Rauf Aliev

Contents

Abstract

This book serves as an essential practitioner's guide to the world of recommender algorithms as it stands in early 2026. We begin with the indispensable baselines—from classic neighborhood models to powerful matrix factorization—and build toward the sophisticated deep learning architectures that power today's largest platforms, including hybrids for CTR prediction and state-of-the-art sequential models.

A core theme of this guide is the practical integration of the latest technological breakthroughs. We dedicate significant attention to the transformative impact of Large Language Models (LLMs), offering architectural blueprints for leveraging them as powerful semantic feature extractors, building reliable Retrieval-Augmented Generation (RAG) pipelines, and designing the next wave of generative and conversational recommender agents. Furthermore, we explore the critical role of multimodal models like CLIP for solving visual cold-start problems and provide insights into specialized areas like debiasing and fairness.

This is more than a survey; it is a toolkit for the modern engineer. Each section balances conceptual depth with pragmatic advice on implementation, scalability, and production readiness, making it the definitive resource for professionals tasked with creating value through personalization.

Introduction

In the modern digital ecosystem, users are confronted with a virtually infinite selection of items, from products and movies to news articles and music. This phenomenon, often termed "information overload," presents a significant challenge for both consumers and platforms. Recommender systems have emerged as a critical technology to address this challenge, serving as personalized information filters that guide users toward relevant content, thereby enhancing user experience, engagement, and commerce.

The field of recommendation algorithms has undergone a remarkable evolution. Early systems were built on simple statistical methods that leveraged direct user-item interactions. These foundational techniques, known as collaborative filtering, gave way to more sophisticated latent factor models, which sought to uncover the hidden dimensions of user preference by decomposing the user-item interaction matrix. The deep learning revolution subsequently ushered in a new era, with neural networks enabling the modeling of complex, non-linear relationships that were previously intractable.

This progression continued with the development of specialized architectures to capture the sequential dynamics of user behavior, borrowing heavily from advances in natural language processing. Concurrently, a new perspective emerged that modeled the recommendation problem as a graph, applying Graph Neural Networks to capture high-order relationships between users and items. Most recently, the landscape is being reshaped by the advent of large-scale generative models, including Generative Adversarial Networks, Diffusion Models, and, most notably, Large Language Models (LLMs), which are redefining the boundaries of what recommender systems can achieve.

This book aims to provide a structured, high-level, and practical overview of this algorithmic landscape. We organize this guide into nine principal chapters based on the primary data modality and methodological approach each class of algorithms leverages:

1. **Foundational and Heuristic-Driven Algorithms.** Models that rely on intrinsic item attributes (Content-Based) or manually defined heuristics (Rule-Based) to generate recommendations, offering interpretability and effectiveness for cold-start scenarios.

2. **Interaction-Driven Recommendation Algorithms.** The core of collaborative filtering, where models rely exclusively on user-item interaction data (e.g., ratings, clicks, purchases).

3. **Context-Aware Recommendation Algorithms.** Advanced models that leverage explicit side features and contextual information, crucial for industrial applications like CTR prediction.

4. **Text-Driven Recommendation Algorithms.** Models that incorporate unstructured text, such as user reviews or item descriptions, and are increasingly powered by LLMs.

5. **Multimodal Recommendation Algorithms.** Models that fuse information from multiple sources, such as text and images, to create a holistic understanding of items and preferences.

6. **Knowledge-Aware Recommendation Algorithms.** Advanced models that leverage structured knowledge from external sources like knowledge graphs.

7. **Specialized Recommendation Tasks.** A look at crucial sub-fields like ensuring fairness, mitigating bias, and addressing the cold-start problem.

8. **New Algorithmic Paradigms.** An exploration of emerging paradigms that extend beyond traditional recommendation, focusing on long-term value, causality, and transparency.

9. **Evaluating Recommender Systems.** A practical guide to the metrics and methodologies used to measure the performance and quality of recommender systems.

For each algorithm, we provide a concise explanation of its core concept, key differentiators, primary use cases, and practical considerations for implementation, along with a link to its seminal paper. Our objective is to equip engineers and researchers with a comprehensive map to navigate the field, understand its historical trajectory, and make informed decisions when designing and deploying the next generation of recommender systems.

Structure and Prerequisites

This book is structured to guide you progressively from foundational concepts to the state-of-the-art, with a focus on practical understanding and implementation. Below is an overview of the book's organization and the knowledge recommended to get the most out of it.

How This Book Is Organized

We have organized the content into thematic sections that build upon one another, guiding you progressively from foundational concepts to the state-of-the-art. This structure is based on the primary data modality and algorithmic approach each family of models utilizes.

- **Chapter 1: Foundational and Heuristic-Driven Algorithms.** We begin with the essential building blocks of recommendation. This includes content-based filtering methods like TF-IDF and the Vector Space Model, alongside simple yet crucial rule-based systems such as "Top Popular". These models serve as powerful baselines and are often vital components in more complex hybrid systems.

- **Chapter 2: Interaction-Driven Recommendation Algorithms.** As the largest and most comprehensive section, this chapter forms the core of both classic and modern collaborative filtering. We journey from traditional neighborhood-based methods (ItemKNN, UserKNN) to the powerful latent-factor models of matrix factorization (SVD, BPR, WRMF). From there, we transition into the deep learning era, covering hybrid architectures (NCF, NeuMF), sophisticated sequential models that learn from user behavior over time (GRU4Rec, SASRec, BERT4Rec), and conclude with cutting-edge generative paradigms like IRGAN and DiffRec.

- **Chapter 3: Context-Aware Recommendation Algorithms.** This chapter focuses on advanced models crucial for industrial applications like CTR prediction, leveraging explicit side features and contextual information (e.g., Wide & Deep, LightGBM).

- **Chapter 4: Text-Driven Recommendation Algorithms..** Here, we explore how to leverage unstructured text, covering models that learn from reviews and item descriptions, followed by a deep dive into the impact of Large Language Models (LLMs).

- **Chapter 5: Multimodal Recommendation Algorithms.** We then expand into the visual domain, discussing how models like CLIP fuse information from text and images, effectively addressing cold-start problems.

- **Chapter 6: Knowledge-Aware Recommendation Algorithms.** This chapter delves into models that leverage external structured knowledge, typically from knowledge graphs, often using Graph Neural Networks (e.g., NGCF, LightGCN).

- **Chapter 7: Specialized Recommendation Tasks.** We then cover focused but essential sub-fields like debiasing, fairness, cross-domain recommendation, and meta-learning for cold-start scenarios.

- **Chapter 8: New Algorithmic Paradigms.** This chapter explores emerging approaches that extend beyond traditional recommendation, such as Reinforcement Learning, Causal Inference, and Explainable AI.

- **Chapter 9: Evaluating Recommender Systems.** The final chapter provides a practical guide to the metrics and methodologies used to measure the performance and quality of recommender systems, covering both rating prediction and ranking tasks.

- **Appendix.** For readers keen on implementation, the appendix provides detailed, line-by-line explanations of the code presented in the tutorials, clarifying the role of each function and library.

Essential Mathematical Basics for Understanding Latent Matrix Models

We assume that the reader is already familiar with all of the following, but for the sake of completeness and consistency of presentation, we provide a very brief introduction to the mathematical foundations that will be frequently referenced in the later chapters of the book.

Matrix Representation in Recommendations

A matrix is a structured grid of numbers, often used in recommendation systems to represent interactions between users and items. Specifically:

- Rows correspond to users (e.g., individuals in a dataset).

- Columns correspond to items (e.g., movies, products).

- Each cell contains an interaction value, such as a rating (e.g., 5 stars) or a binary indicator (1 for interaction, 0 otherwise).

When most cells are empty—because users interact with only a small subset of items—the matrix is described as sparse. Latent matrix models aim to predict these missing values by uncovering patterns within the observed data.

Example — A small matrix R with 3 users and 4 items might appear as:

$$R = \begin{pmatrix} 5 & ? & 3 & ? \\ ? & 4 & ? & 2 \\ 1 & ? & 5 & ? \end{pmatrix}$$

Here, "?" denotes unknown values to be predicted.

Vectors and Dot Products

A vector is a sequence of numbers, such as a row or column extracted from a matrix. In latent matrix models, users and items are represented by *latent vectors*—compact lists of numbers capturing hidden characteristics (e.g., a user's preference for action or romance genres, inferred from data).

The dot product measures the similarity between two vectors by computing a single scalar value. For vectors $\vec{u} = (u_1, u_2, \ldots, u_k)$ and $\vec{i} = (i_1, i_2, \ldots, i_k)$, the dot product is defined as:

$$\vec{u} \cdot \vec{i} = u_1 i_1 + u_2 i_2 + \cdots + u_k i_k$$

In recommendation systems, the dot product between a user's latent vector and an item's latent vector predicts the expected rating, with a higher value indicating a stronger predicted preference.

Example — A user's latent vector (0.8 for action, 0.2 for romance) and an item's vector (0.9 for action, 0.1 for romance) yield a dot product of $0.8 \times 0.9 + 0.2 \times 0.1 = 0.74$, suggesting a likely positive match.

Matrix Factorization (Decomposition)

Matrix factorization involves decomposing a large user-item matrix R into two smaller matrices, U (containing user latent vectors) and V (containing item latent vectors), such that their product approximates R:

$$R \approx U \times V^T$$

Here, V^T denotes the transpose of V, aligning item vectors as columns. The term *latent* refers to the hidden factors (e.g., genres, styles) that these vectors represent, learned directly from the data.

This decomposition is efficient because it reduces dimensionality. For a matrix R with m users and n items, U is $m \times k$ and V is $n \times k$, where k (the number of latent factors) is small (e.g., 10–100). This approach addresses sparsity by predicting missing entries based on learned patterns.

Loss Functions and Optimization

Latent matrix models optimize U and V by minimizing a loss function, which quantifies the discrepancy between predicted and actual ratings. A common choice is the Mean Squared Error (MSE), defined for actual ratings r and predicted ratings $\hat{r}$ as:

$$\text{MSE} = \frac{1}{N} \sum (r - \hat{r})^2$$

Optimization techniques, such as gradient descent, iteratively adjust U and V to reduce this error. Gradient descent starts with random U and V and updates them in small steps to approach the minimum loss, analogous to finding the lowest point in a landscape.

Some models, like ALS, use an alternating approach, where U is fixed to optimize V, then V is fixed to optimize U, repeating until convergence.

Regularization

To prevent overfitting—where models memorize noise rather than generalizing patterns—regularization is employed. This technique penalizes overly large values in U and V, encouraging simpler solutions.

A standard regularization method is L2 regularization, which adds a penalty term to the loss function:

$$\lambda(|\vec{u}|^2 + |\vec{i}|^2)$$

Here, λ controls the strength of the penalty, and $|\vec{u}|^2$ and $|\vec{i}|^2$ represent the squared magnitudes of the user and item vectors, respectively. This ensures the model focuses on meaningful patterns.

Overview of Recommender Algorithms

Foundational and Heuristic-Driven Algorithms

Vector Space Model (VSM) — A foundational framework for content-based filtering that represents items and user preferences as vectors in a high-dimensional feature space, where similarity is calculated based on vector proximity. (see page 17)

TF-IDF — A classic weighting scheme for the Vector Space Model (VSM) that calculates feature weights based on a term's frequency within an item and its inverse frequency across the entire catalog, making it an effective baseline for recommending new items with descriptive metadata. (see page 21)

Embedding-based Similarity (Word2Vec) — This latent approach to VSM uses models like Word2Vec to learn dense feature embeddings based on co-occurrence, capturing semantic relationships that TF-IDF misses. The model is trained to predict context features that appear alongside a given feature, creating a vector space where similar features are mapped closely together. (see page 22)

CBOW (Continuous Bag-of-Words) — As an alternative to Word2Vec's Skip-gram, the CBOW model learns feature embeddings by predicting a central feature from the average of its surrounding context features, offering a more computationally efficient approach that performs well for frequent features. (see page 37)

FastText — Extending Word2Vec, FastText addresses the out-of-vocabulary problem by learning embeddings for character n-grams rather than entire features, allowing it to construct meaningful vectors for rare or previously unseen features by combining the representations of their constituent parts. (see page 38)

Classic Rule-Based Systems — These systems use manually defined, domain-specific logical rules (e.g., "if-then") to generate recommendations based on business knowledge rather than statistical learning, offering complete transparency and control but lacking personalization and scalability. (see page 39)

Top Popular — This simple heuristic-driven algorithm recommends items based on their overall popularity across all users, typically measured by the total number of interactions, serving as a common non-personalized baseline that is effective for the cold-start problem. (see page 40)

Apriori — the classic algorithm for association data mining. It works iteratively by first identifying individual items that meet a minimum support threshold (i.e., appear frequently). It then uses these frequent items to generate candidate pairs, tests their frequency, and continues this level-by-level process for triplets, quadruplets, and so on. Its efficiency comes from the **Apriori Principle**: if an itemset is frequent, all of its subsets must also be frequent, which allows the algorithm to prune a vast number of candidates early on. However, its primary drawback is the need to make multiple passes over the database and the potential to generate a massive number of candidate sets, making it slow for large datasets. (see page 41)

FP-Growth — more efficient alternative that avoids the costly candidate generation step of Apriori. It works by first compressing the entire transactional database into a compact tree structure known as an FP-tree. This tree stores the itemset information in a way that allows the algorithm to mine frequent itemsets directly from this structure with just two passes over the data. This makes it significantly faster and more memory-efficient, especially for large datasets. (see page 44)

Eclat — takes a different approach by using a **vertical data layout**. Instead of listing items for each transaction (horizontal format), it creates a list of transaction IDs for each item. It then finds frequent itemsets by simply intersecting these transaction ID lists. (see page 45)

Interaction-Driven Recommendation Algorithms

ItemKNN — A classic neighborhood-based collaborative filtering model that recommends items similar to those a user has previously interacted with. It computes item-to-item similarity based on the interaction patterns of all users, making it more scalable than user-based approaches when the number of users greatly exceeds the number of items. (see page 53)

SAR (Smart Adaptive Recommendations) — A fast and scalable item-based model optimized for implicit feedback that builds an item similarity matrix based on co-occurrence counts rather than more complex metrics like cosine similarity, making it a simple yet powerful baseline. (see page 64)

UserKNN — In contrast to ItemKNN, this model finds users with similar interaction histories to a target user and recommends items that those ”neighboring” users liked. This approach often generates more diverse and serendipitous recommendations but is less scalable as the user base grows. (see page 74)

SlopeOne — An extremely simple item-based algorithm that predicts ratings based on the pre-computed average rating *deviation* between pairs of items. It has no iterative training phase and can be updated incrementally as new ratings arrive, making it lightweight and fast. (see page 81)

Attribute-Aware k-NN — An extension of k-NN that combines both collaborative signals (user interactions) and content-based signals (item attributes) into a hybrid similarity metric, which is particularly effective for alleviating the item cold-start problem. (see page 84)

FunkSVD (Biased Matrix Factorization) — A breakthrough matrix factorization model that predicts ratings as the sum of a global average, user and item biases, and the dot product of user and item latent vectors. Unlike true SVD, it is an optimization-based model that works on sparse data and learns its parameters via methods like Stochastic Gradient Descent. (see page 91)

PMF (Probabilistic Matrix Factorization) — A probabilistic version of matrix factorization that treats latent factors as probability distributions and assumes ratings are drawn from a Gaussian distribution centered at the user-item dot product, providing a principled Bayesian approach to regularization. (see page 96)

WRMF (Weighted Regularized Matrix Factorization) — A matrix factorization model specifically designed for implicit feedback, which treats all unobserved items as negative examples but assigns them a much lower confidence weight than positive interactions. It is typically optimized with an efficient Alternating Least Squares (ALS) algorithm. (see page 103)

BPR (Bayesian Personalized Ranking) — A foundational model that reframes recommendation from rating prediction to a pairwise ranking task. It is trained to ensure that for any user, a positive (interacted-with) item has a higher prediction score than a negative (unobserved) item, making it ideal for optimizing ranked lists from implicit feedback. (see page 111)

SVD++ — An extension of FunkSVD (Biased MF) that incorporates implicit feedback to enrich the user's latent representation. A user's preference is modeled not just by their explicit latent vector but also by an aggregated vector representing all the items they have interacted with, regardless of the rating. (see page 114)

TimeSVD++ — An extension of SVD++ that incorporates temporal dynamics by modeling user biases, item biases, and user factors as functions of time, allowing it to capture the dynamic nature of preferences and item popularity. (see page 123)

SLIM & FISM — These models directly learn a sparse item-to-item similarity matrix from implicit data. SLIM uses a sparse linear model with L1/L2 regularization to create an efficient and interpretable matrix, while FISM factorizes the similarity matrix into two smaller embedding matrices to better capture latent relationships and handle extreme data sparsity. (see page 125)

Non-Negative Matrix Factorization (NonNegMF) — A variant of matrix factorization that constrains the user and item latent factors to be non-negative, resulting in an additive, parts-based representation that is often more interpretable than standard matrix factorization. (see page 136)

CML (Collaborative Metric Learning) — Reframes collaborative filtering as a metric learning problem, learning a shared embedding space where the Euclidean distance between a user and item reflects their compatibility. It uses a pairwise loss to pull positive user-item pairs closer together than negative pairs, serving as a powerful alternative to BPR for ranking tasks. (see page 140)

NCF (Neural Collaborative Filtering) & NeuMF — NCF is a framework that generalizes matrix factorization by replacing the dot product with a flexible multi-layer perceptron (MLP) to capture complex, non-linear user-item interactions. Its flagship model, NeuMF, fuses a linear matrix factorization path with a non-linear MLP path to combine the strengths of both approaches for state-of-the-art performance on implicit feedback tasks. (see page 144)

DeepFM & xDeepFM — These are hybrid architectures for CTR prediction that combine a ”wide” component for memorizing low-order feature interactions with a ”deep” component for generalizing to high-order interactions. DeepFM uses a Factorization Machine for the wide part, while xDeepFM enhances this with a Compressed Interaction Network (CIN) to learn explicit, high-order feature interactions in a more controlled manner. (see page 150)

Autoencoder-based (DAE & VAE) — This approach frames recommendation as a reconstruction task, training a neural network to compress a user's entire interaction history into a low-dimensional vector and then reconstruct it. Denoising Autoencoders (DAEs) learn robust representations from corrupted inputs, while Variational Autoencoders (VAEs) offer a probabilistic, generative approach that is particularly effective for modeling the uncertainty in implicit feedback data. (see page 154)

SimpleX — A simple yet powerful bi-encoder baseline for collaborative filtering that achieves state-of-the-art performance through a contrastive learning objective (InfoNCE). It learns to pull positive user-item pairs together in an embedding space while pushing them apart from in-batch negative items, enhanced by consistency regularization for more robust training. (see page 160)

EASE (Embarrassingly Shallow Autoencoders) — A non-neural, linear autoencoder that learns an item-item similarity matrix to reconstruct user interaction histories. Its key differentiator is a closed-form analytical solution that eliminates the need for iterative training, making it an extremely strong, simple, and reproducible baseline for implicit feedback tasks. (see page 163)

GRU4Rec — A pioneering sequential model that applies a Gated Recurrent Unit (GRU), a type of RNN, to session-based recommendation. It processes a sequence of user interactions one by one, maintaining an evolving hidden state that captures the user's current intent to predict the next item they will interact with. (see page 164)

NextItNet — An alternative sequential model that uses stacked layers of dilated 1D convolutions to efficiently identify patterns and long-range dependencies in user interaction histories. Unlike RNNs, its convolutional architecture can process all parts of a sequence in parallel, making training significantly faster and more scalable for very long sequences. (see page 167)

SASRec & BERT4Rec — These models apply the Transformer architecture's self-attention mechanism to sequential recommendation. SASRec is unidirectional, using masked self-attention to predict the next item based only on past actions. In contrast, BERT4Rec is bidirectional and is trained to predict masked items using both past and future context, allowing it to learn a richer, more holistic representation of user interests. (see page 170)

CL4SRec (Contrastive Learning for Sequential Recommendation) — A framework that enhances sequential models like SASRec by adding an auxiliary contrastive learning objective. It trains the model to recognize that two different augmented views of the same user sequence should have similar representations, forcing it to learn more robust and generalizable user profiles, especially in sparse data scenarios. (see page 173)

IRGAN — Adapts the Generative Adversarial Network (GAN) framework to recommendation by training a Generator (the recommender) to produce plausible user-item pairs and a Discriminator to distinguish them from real interactions. This adversarial game forces the Generator to learn a more robust model of user preferences, effectively performing a type of hard negative mining. (see page 177)

DiffRec — A generative model that adapts Denoising Diffusion Probabilistic Models (DDPMs) to recommendation. It learns the distribution of user preferences by training a neural network to reverse a fixed process of gradually adding noise to user interaction profiles, enabling the generation of high-quality and diverse recommendations from pure noise. (see page 180)

GFN4Rec — Frames recommendation as a sequential list-building task using Generative Flow Networks (GFlowNets). The model learns a policy to construct a slate of items step-by-step, where the probability of generating a list is proportional to a predefined reward, inherently promoting both relevance and diversity in the final set of recommendations. (see page 183)

IDNP (Interest Dynamics Neural Process) — Leverages the principles of Normalizing Flows, a class of generative models that learn a data distribution through a series of invertible transformations. This allows it to model a user's preference as a complex probability distribution over time, capturing uncertainty and making it particularly promising for few-shot or cold-start sequential recommendation tasks. (see page 186)

Text-Driven Recommendation Algorithms

DeepCoNN — A foundational review-based model that uses two parallel Convolutional Neural Networks (CNNs) to learn latent representations directly from raw text. One CNN processes all reviews written *by* a user to form a user vector, while the other processes all reviews written *for* an item to form an item vector, which are then combined to predict a rating. (see page 205)

NARRE — An extension of DeepCoNN that incorporates a dual attention mechanism to identify and assign higher weights to the most useful and informative reviews when constructing user and item representations. This not only improves accuracy by filtering out noise but also provides a natural pathway to explainability by highlighting the most influential reviews. (see page 209)

LLM-based Retrieval (Dense Retrieval & Cross-Encoders) — This is a standard two-stage architecture for large-scale systems. First, a fast bi-encoder independently creates embeddings for a user query and all items to retrieve a set of candidates from a massive catalog. Second, a slower but more accurate cross-encoder re-ranks these candidates by processing the user query and each item *together* to produce a precise relevance score. (see page 214)

LLM-based Generative / Instruction-Tuned — This approach reframes recommendation as a text generation task where an instruction-tuned LLM is given a prompt with a user's history and a task (e.g., ”Recommend 5 sci-fi movies and explain why”). The model leverages its vast world knowledge and zero-shot reasoning ability to generate a coherent, natural language response containing both recommendations and justifications. (see page 218)

LLM-based RAG & Feature Extraction — This paradigm uses an LLM as a powerful component to enhance other systems. It can act as a sophisticated feature extractor, creating high-quality semantic embeddings from text to feed into any downstream model. Alternatively, in a Retrieval-Augmented Generation (RAG) setup, it grounds a generative LLM with externally retrieved, up-to-date information to ensure its recommendations are factually accurate and to mitigate hallucinations. (see page 222)

LLM Agents & Tool Use — An advanced paradigm where the LLM acts as a reasoning engine that can autonomously plan and use external tools (e.g., APIs, databases, other recommender models). It decomposes a complex user request into a series of sub-tasks and tool calls to fulfill multi-step, sophisticated goals beyond what a single model can achieve. (see page 222)

Conversational RecSys (Dialogue-based) — This approach creates an interactive, multi-turn dialogue to actively elicit a user's preferences, especially in cold-start scenarios. The system acts like a guided assistant, asking clarifying questions to progressively narrow down options and understand the user's true, often ambiguous, intent. (see page 223)

Multimodal Recommendation Algorithms

CLIP (Contrastive Language-Image Pre-Training) — A foundational model that learns a shared embedding space for images and text by training on a massive dataset of (image, caption) pairs with a contrastive objective. Its powerful zero-shot transfer capability allows it to understand visual concepts described in text without explicit fine-tuning, making it an extremely effective tool for generating semantic embeddings from item images to solve the visual cold-start problem. (see page 226)

ALBEF (Align Before Fuse) — A multimodal architecture that first aligns image and text features from unimodal encoders using a contrastive loss, and *then* fuses them using a cross-modal encoder. Its multi-task training objective allows it to learn finer-grained interactions between visual regions and words, making it a state-of-the-art model for complex multimodal reasoning tasks. (see page 228)

Multimodal VAEs — Extends the Variational Autoencoder framework to learn a joint latent probability distribution across multiple modalities (e.g., image and text). Its probabilistic nature makes it excellent at handling missing modalities and enables cross-modal generation, such as creating a likely text description for a given product image. (see page 232)

Multimodal Diffusion — Applies the denoising diffusion process to multimodal data, learning the joint distribution by training a model to reverse the process of gradually adding noise to data from multiple modalities simultaneously. This paradigm is capable of generating exceptionally high-quality and realistic multimodal content, representing the cutting edge of generative modeling. (see page 233)

Context-Aware Recommendation Algorithms

Factorization Machines (FM) — A powerful model for context-aware recommendation that extends matrix factorization to handle any real-valued feature vector. Its key innovation is efficiently modeling all pairwise feature interactions using factorized parameters, which allows it to estimate interaction strengths for feature combinations it has never seen in the training data, thus solving the data sparsity problem. (see page 191)

AMF (Attentional Factorization Machine) — An extension of Factorization Machines that incorporates an attention mechanism to learn the importance of different pairwise feature interactions. This allows the model to focus on the most predictive combinations while filtering out noise, improving both accuracy and interpretability for CTR prediction tasks. (see page 193)

Wide & Deep — A pioneering hybrid architecture from Google that jointly trains a "wide" linear component for memorizing frequent feature co-occurrences and a "deep" neural network for generalizing to unseen feature combinations. This unified approach allows the model to be both powerful and efficient, making it a foundational architecture for large-scale industrial CTR prediction. (see page 194)

LightGBM — A high-performance gradient boosting framework that uses an ensemble of decision trees to predict CTR based on a wide array of user, item, and contextual features. Its primary differentiators are its speed and efficiency, which are achieved through leaf-wise tree growth and native handling of categorical features, making it a workhorse for industrial ranking tasks on tabular data. (see page 198)

Knowledge-Aware Recommendation Algorithms

NGCF (Neural Graph Collaborative Filtering) — A pioneering model that applies a standard Graph Neural Network (GNN) to the user-item interaction graph. It learns embeddings by iteratively aggregating features from a node's neighborhood, including feature transformation matrices and non-linear activations, to capture high-order connectivity patterns. (see page 235)

LightGCN — A radical simplification of NGCF that achieves superior performance by removing the feature transformation matrices and non-linear activations, leaving only the core GNN component of neighborhood aggregation. It learns user and item embeddings by simply taking the weighted sum of their neighbors' embeddings from the previous layer, proving that for CF, effectively modeling the graph structure is key. (see page 236)

SGL (Self-supervised Graph Learning) — A training paradigm applied on top of GNN-based recommenders like LightGCN to improve embedding quality. It introduces an auxiliary contrastive learning task, training the model to recognize that a node's representation should be similar across two different augmented views of the graph, which forces it to learn more robust and generalizable representations, especially in sparse data scenarios. (see page 238)

New Algorithmic Paradigms

Reinforcement Learning (RL) for RecSys — This paradigm frames recommendation as a sequential decision problem where an "agent" (the system) learns a "policy" to recommend items that maximize a long-term reward, such as user engagement over an entire session. It shifts the focus from predicting a single accurate interaction to optimizing the entire sequence of recommendations for long-term value. (see page 245)

Causal Inference in RecSys — This approach moves beyond correlation to understand causation, aiming to recommend items that will actually *persuade* a user to interact, rather than just recommending items they were likely to choose anyway. Techniques like Uplift Modeling identify "persuadable" users to maximize the true incremental impact of the recommendations. (see page 246)

Explainable AI (XAI) for RecSys — This field focuses on making the recommendation process transparent and trustworthy by answering the question "Why was this recommended to me?". Methods include generating explanations by finding paths in a knowledge graph or highlighting the most influential user/item features that led to the recommendation. (see page 247)

1 Foundational and Heuristic-Driven Algorithms

Before the dominance of collaborative filtering, the recommendation landscape was shaped by foundational approaches that remain relevant today as powerful baselines, components of hybrid systems, and transparent business logic engines. These algorithms operate not on user interaction patterns but on intrinsic item attributes (Content-Based) or manually defined heuristics (Rule-Based). They are highly interpretable, easy to implement, and effectively address the cold-start problem for new items.

1.1 Content-Based Filtering

Content-based filtering recommends items similar to those a user has liked in the past, **based on the attributes of the items themselves**. It operates on the principle: "Show me more of what I like," by building a user profile from their interactions (e.g., likes, ratings, or views) and matching it to item features. This approach is independent of other users' data, making it robust against data sparsity and effective for recommending niche items or new items with descriptive metadata. However, it may over-specialize, recommending only highly similar items, and relies heavily on high-quality metadata.

> **How it works...** Imagine you've watched and loved the movie *Blade Runner*. A content-based system would analyze its attributes: `Genre: Sci-Fi`, `Theme: Cyberpunk`, `Director: Ridley Scott`. It would then search its catalog for other items with similar attributes, such as *Alien* (also directed by Ridley Scott) or *Ghost in the Shell* (also sci-fi and cyberpunk), and recommend them to you. The user's taste is built as a profile based on the content of items they've previously enjoyed.

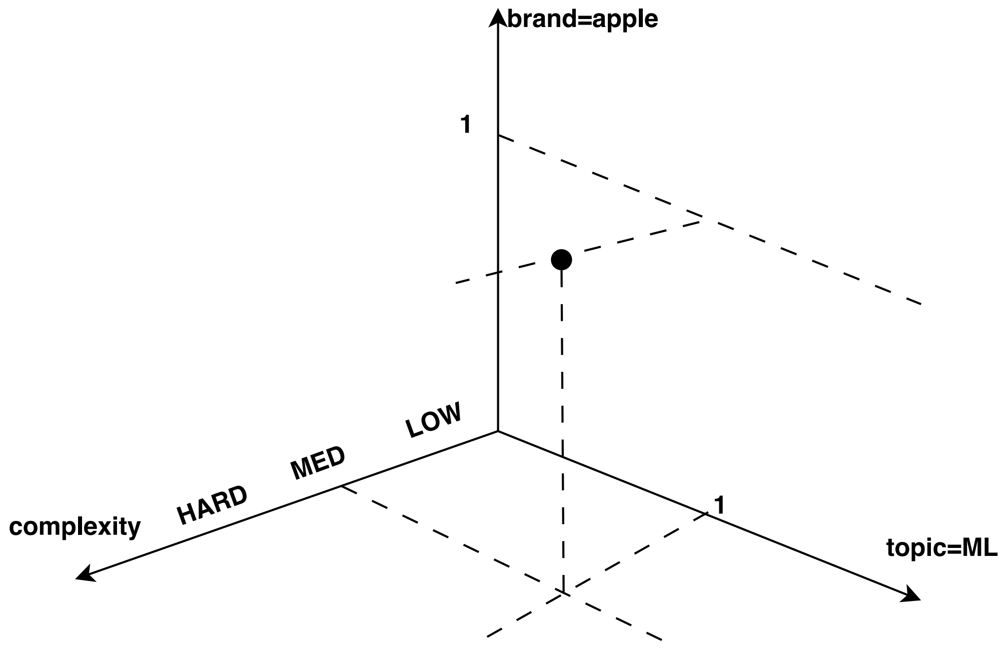
1.1.1 The Vector Space Model (VSM): A Foundational Framework

Key concept — The Vector Space Model (VSM) is the foundational framework for content-based filtering. It represents items and user preferences as vectors in a high-dimensional space where each dimension corresponds to a specific feature (e.g., a genre, a brand, a keyword). Similarity between items, or between an item and a user, is then calculated as the proximity of their vectors in this space.

Key differentiator — VSM is not a single algorithm but a flexible **model** that provides the theoretical underpinning for many content-based techniques. It separates the abstract idea of vector representation from the specific implementation of how vector weights are calculated (e.g., TF-IDF or binary), making it a versatile and fundamental concept.

Use cases — As a general framework, it's the basis for recommending articles, products, or media based on metadata. It is particularly effective for explaining the mechanics of content-based filtering and for building systems in cold-start scenarios where item attributes are available but interaction data is sparse.

When to Consider — Understanding VSM is the first step in building any content-based recommender. It provides the essential blueprint for representing content numerically, upon which specific weighting schemes and similarity metrics are built.



Seminal Papers:

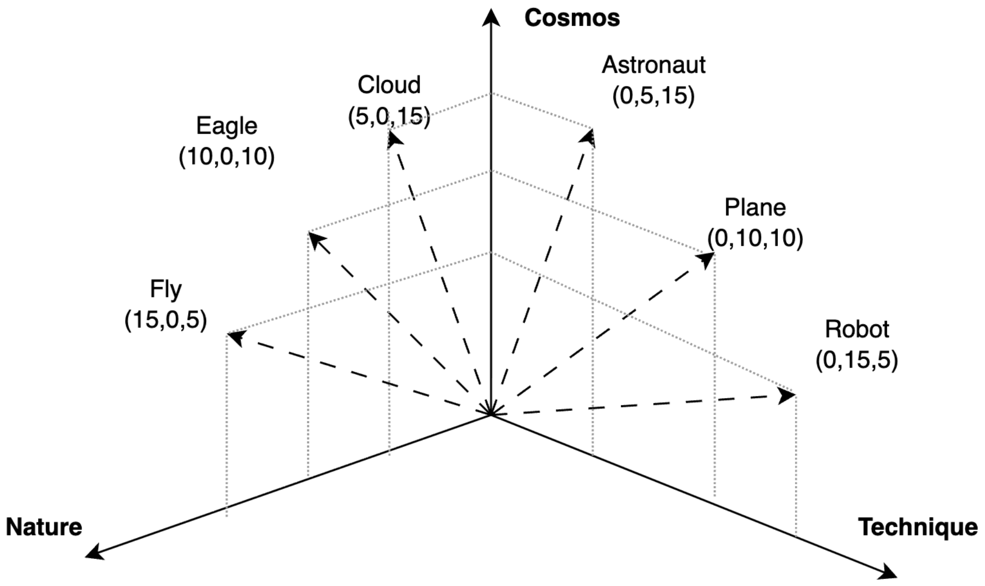
- Salton, G., Wong, A., & Yang, C. S. (1975). *A Vector Space Model for Automatic Indexing*. https://dl.acm.org/doi/10.1145/361219.361220.

Useful links:

- Vector Space Model, Wikipedia article [https://en.wikipedia.org/wiki/Vector_space_model]

Mathematical Foundations of VSM

The Vector Space Model is an algebraic framework that allows us to perform geometric and mathematical operations on items by representing them as numerical vectors.



To generalize from its origins in text analysis, we map:

- A "document" is an **item** in the catalog.
- A "term" is a **feature** (e.g., `genre='sci-fi'`, `brand='Apple'`, `topic='machine-learning'`).
- The "corpus" is the entire **catalog of items**.

Part A: Representing Items as Vectors

The first step is to define the dimensions of the vector space by creating a "vocabulary" of all unique features across the entire catalog, $F = \{f_1, f_2, \ldots, f_n\}$. Each item i is then represented as an n-dimensional vector $\vec{i}$:

$$\vec{i} = (w_{1,i}, w_{2,i}, \ldots, w_{n,i})$$

The value $w_{k,i}$ is a **weight** that signifies the importance of feature f_k in item i.

Part B: Feature Weighting Schemes

The VSM framework supports various methods for calculating the weight $w_{k,i}$. The choice of scheme is crucial for performance. Common schemes include:

- **Binary:** A simple scheme where $w_{k,i} = 1$ if feature f_k is present in item i, and 0 otherwise.
- **Term Frequency (TF):** The weight is the count or normalized frequency of the feature in the item.
- **TF-IDF:** The most common and effective scheme, which assigns a high weight to features that are important to a specific item but rare across the entire catalog. We will detail its calculation in the next section.

Part C: Measuring Similarity with Cosine Similarity

Once items are represented as vectors, their similarity is measured by their proximity. **Cosine Similarity** is the standard metric, measuring the cosine of the angle between two vectors. It is robust to differences in the number of features (vector magnitude) and effectively captures similarity. For two item vectors, $\vec{A}$ and $\vec{B}$:

$$\text{similarity}(\vec{A}, \vec{B}) = \cos(\theta) = \frac{\vec{A} \cdot \vec{B}}{|\vec{A}||\vec{B}|} = \frac{\sum_{k=1}^{n} A_k B_k}{\sqrt{\sum_{k=1}^{n} A_k^2}\sqrt{\sum_{k=1}^{n} B_k^2}}$$

A score near **1** indicates high similarity, while a score near **0** indicates dissimilarity.

Part D: Creating User Profiles for Recommendation

VSM can be extended to recommend items to a user by creating a vector representation for that user's preferences. A user's profile vector, $\vec{u}$, is computed by aggregating the vectors of the items they have positively interacted with (e.g., rated highly, purchased). A common method is to calculate the average of these item vectors:

$$\vec{u} = \frac{1}{|I_u|} \sum_{i \in I_u} \vec{i}$$

where I_u is the set of items the user liked. Recommendations are then generated by calculating the cosine similarity between the user's profile vector $\vec{u}$ and all unseen item vectors.

If a user u is new, the set I_u is empty, and you cannot compute a profile vector for them (so called a user cold-start problem).

1.1.2 TF-IDF: A Classic VSM Weighting Scheme

Key concept — This classic approach provides a specific method for calculating the feature weights within the Vector Space Model. **TF-IDF (Term Frequency-Inverse Document Frequency)** converts an item's features into a numerical vector, assigning scores that reflect each feature's importance to the item relative to the entire catalog.

Key differentiator — Its simplicity, interpretability, and effectiveness as a baseline. Unlike collaborative filtering, it can recommend brand-new items as long as they have descriptive features, solving the item cold-start problem. However, it assumes feature independence, which may miss semantic relationships (e.g., "sci-fi" and "space opera" being related).

Use cases — Recommending articles based on topics, suggesting products based on specifications, or finding similar movies based on genres and directors. It's ideal for domains with rich metadata.

When to Consider — Use TF-IDF as a first-pass content-based recommender or a strong baseline. It excels when you need a fast, simple, and explainable model that relies purely on item features, especially with diverse descriptive features.

Mathematical Foundations

This approach provides a precise formula for the feature weights, $w_{k,i}$, used in the Vector Space Model.

Part A: Calculating the TF-IDF Score

The TF-IDF score reflects a feature's importance to an item within the catalog and combines two components:

(1) **Term Frequency (TF):** Measures how relevant a feature f is to an item i. For non-text features, it's typically binary (1 if present, 0 otherwise) or a weighted relevance score. A common normalized form is:

$$\text{tf}(f, i) = \frac{\text{count or relevance of feature } f \text{ in item } i}{\text{total number of features in item } i}$$

(2) **Inverse Document Frequency (IDF):** Measures the feature's rarity across the entire catalog I, giving higher weight to rare features and lower weight to common ones. To avoid division by zero, smoothing is applied:

$$\text{idf}(f, I) = \log\left(\frac{|I| + 1}{\text{Number of items with feature } f + 1}\right)$$

The final **TF-IDF score** for a feature is the product of these two values, representing its weight in the item's vector:

$$\text{score}(f, i, I) = \text{tf}(f, i) \times \text{idf}(f, I)$$

Because any single item only has a small subset of all possible features, these vectors are typically very sparse.

Part B: Making a Recommendation

(1) For each item in the catalog, construct its vector using the TF-IDF scores of its features.

(2) Take a target item (e.g., the last item a user interacted with).

(3) Calculate the Cosine Similarity between its TF-IDF vector and all other items' vectors.

(4) Recommend the items with the highest similarity scores.

1.1.3 Embedding-based Similarity: A Latent Approach to VSM

These models learn dense embeddings using the distributional hypothesis: ”features that appear in similar contexts (e.g., co-occurring in the same items) have related meanings.” In recommendation systems, this hypothesis extends to non-text domains like item tags, where features such as ”sci-fi” and ”cyberpunk” co-occurring in movies suggest semantic similarity. By learning latent representations, these embeddings capture underlying relationships not explicitly defined in metadata, making them a powerful approach to content-based filtering. A neural network learns these embeddings via a pretext task.

Imagine training a model like a game of word association, but for features—given ”sci-fi”, predict ”cyberpunk” but not ”romance”. This contrasts with the Vector Space Model (VSM) described earlier, which relies on deterministic vector weights from explicit statistics (e.g., TF-IDF). Here, the probabilistic nature of embedding models, such as Skip-gram, enables them to capture subtler semantic nuances through learning.

To avoid confusion with VSM, it's important to understand the training mechanism. The described approach, Skip-gram with Negative Sampling, reframes the prediction task as a series of independent binary classification problems. For each positive feature pair, the model learns to distinguish it from several randomly chosen ”negative” pairs. This is achieved using the sigmoid function, which outputs a probability for each individual pair, a process fundamentally different from VSM's algebraic framework. This contrasts with an alternative training method for Word2Vec, Hierarchical Softmax, which does use a Softmax layer to model a probability distribution over the entire feature vocabulary.

The process of using embeddings for recommendations can be broken down into two distinct stages: first, we **train** a model like Skip-gram to learn high-quality vector representations for our features; second, we **use** those representations to calculate similarity and generate a ranked list of recommendations.

Stage 1: Training (Creating a "Map" of Features with Skip-gram)

The first stage is dedicated to building the core components: the feature embeddings. The goal is not to create recommendations directly, but to learn a meaningful "map" where features with similar meanings are located close to each other. The Skip-gram model is an excellent tool for this map-making process.

The objective is to learn a dense vector for each unique feature. The **Skip-gram with Negative Sampling** architecture trains the model to predict context features that co-occur with a center feature, while distinguishing "wrong" combinations. For example, in a movie database, if "sci-fi" is the center feature for *Blade Runner*, context features might include "cyberpunk" or "dystopian", while "romance" (a negative sample) should score low.

In the Skip-gram with Negative Sampling model, used for learning feature embeddings, two key terms come up: **center feature** and **context feature**. These are crucial for understanding how the model learns relationships between features (like tags, words, or attributes) based on their co-occurrence in items (like movies, books, or products).

- **Center Feature (f_c)**: This is the feature the model focuses on at a given moment during training. For example, in a movie database, the center feature might be the tag "sci-fi" from a movie's description. The model systematically iterates through all features in the dataset, selecting one at a time as the center feature to ensure comprehensive learning. The model is trying to figure out which other features (like "cyberpunk" or "action") tend to appear alongside "sci-fi" in the same movie. Understanding means the model learns to predict these related features by adjusting numerical vectors (embeddings) so that features appearing together have similar vectors. For example, if "sci-fi" and "cyberpunk" often show up in the same movies, their vectors become closer in a mathematical "space," capturing their relationship.

- **Context Feature (f_o)**: These are the other features that appear alongside the center feature in the same item. For instance, if "sci-fi" is the center feature for a movie like *Blade Runner*, context features might include "cyberpunk," "dystopian," or "futuristic" because they're also tags for that movie. The model learns to predict these context features based on the center feature, figuring out which ones naturally go together.

The model's objective is to learn to predict "context features" f_o that co-occur with a "center feature" f_c. It's trained to give high similarity scores to true co-occurring pairs (e.g., `('sci-fi', 'cyberpunk')`) and low scores to random "negative" pairs (e.g., `('sci-fi', 'romance')`).

The objective function for a true pair (f_c, f_o) is:

$$\mathcal{J} = \log \sigma(v'_{f_o} \cdot v_{f_c}) + \sum_{i=1}^{k} \mathbb{E}_{f_i \sim P_n(f)}[\log \sigma(-v'_{f_i} \cdot v_{f_c})]$$

Let's break it down:

- $\mathcal{J}$ is the overall objective function for this training example—we aim to maximize it. In practice, optimization algorithms minimize the negative of this expression, $\mathcal{L} = -\mathcal{J}$.
- The first term, $\log \sigma(v'_{f_o} \cdot v_{f_c})$, focuses on the true pair, pushing the model to give them a high similarity score.
- The second term, $\sum_{i=1}^{k} \mathbb{E}_{f_i \sim P_n(f)}[\log \sigma(-v'_{f_i} \cdot v_{f_c})]$, handles the "negative" examples. It pushes the model to give low similarity scores to k randomly sampled features that do not belong with the center feature.
- v_f and v'_f are two vector representations for a feature f.
- $\sigma(\cdot)$ is the sigmoid function, $\sigma(x) = 1/(1 + e^{-x})$.

Key Hyperparameters — The embedding dimension (e.g., 50–300), the number of negative samples k (e.g., 5–20), and the noise distribution $P_n(f)$ significantly affect performance. For item features, the definition of context is also crucial; in our case, we consider all co-occurring features within the same item to be part of the context. The noise distribution $P_n(f)$ is often set as the unigram distribution raised to the 0.75 power, as proposed in the original Word2Vec paper.

At the end of this training stage, the job of Skip-gram is done. The final output is not a list of recommendations, but a high-quality "dictionary" where each feature in our catalog is mapped to a dense vector embedding.

After training, the vectors v_f become the feature embeddings.

Note on Applicability — While this approach is designed for content-based recommendation using item features (e.g., tags or attributes), it can be adapted for collaborative filtering by treating items themselves as "features" and user interaction lists (e.g., rated movies) as "contexts," as shown in the accompanying tutorial code. This creates item embeddings akin to item2vec, blending content-based and collaborative approaches.

References:

- Mikolov, T., Sutskever, I., Chen, K., Corrado, G., & Dean, J. (2013). Distributed Representations of Words and Phrases and their Compositionality. Advances in Neural Information Processing Systems (NIPS). https://arxiv.org/abs/1310.4546

Caveat — Skip-gram with Negative Sampling is efficient but sensitive to noisy data. An alternative, hierarchical softmax, may be considered for smaller datasets or when computational resources are limited, as it avoids sampling negatives but requires a tree structure.

Stage 2: Usage (Generating a Ranked Recommendation List)

Once the "map" of feature embeddings is created, we can use it to plot a route—that is, to generate an ordered list of recommendations for a user. This is a separate, multi-step process that uses the embeddings learned in Stage 1.

Create Item Vectors from Feature Embeddings

Since an item's features are typically an unordered set (e.g., the tags "sci-fi" and "action" are equivalent to "action" and "sci-fi"), the most direct approach is to aggregate their feature embeddings. The simplest and most common aggregation method is **averaging**. The item's vector $\vec{x}_i$ is computed as the element-wise average of the vectors of all its features. This creates a vector representing the item's semantic center of gravity.

$$\vec{x}_i = \frac{1}{N}\sum_{k=1}^{N} v_{f_k}$$

This approach is computationally efficient and correctly treats the features as a "bag of features," ignoring any arbitrary order.

Calculating Similarity

Once all items are represented as dense vectors, we can determine the order of recommendations by measuring proximity. We take a target item vector (e.g., the last item a user liked) and compute the **Cosine Similarity** between its vector and the vectors of all other items in the catalog.

$$\text{similarity}(\vec{x}_i, \vec{x}_j) = \frac{\vec{x}_i \cdot \vec{x}_j}{|\vec{x}_i||\vec{x}_j|}$$

Ranking and Recommending

The final recommendation list is generated by sorting all items in descending order based on their cosine similarity scores. The items with the highest scores are the top recommendations, presented to the user.

This can be done for a target item or for a user's profile vector, which can be created by averaging the vectors of all items they have positively interacted with.

Feature embeddings (Step 1) capture the semantics of individual attributes, enabling fine-grained similarity between tags. Item embeddings (Step 2) build on this by representing entire items, either by aggregating feature embeddings or learning a holistic vector. This shift from feature-level to item-level representations allows recommendations based on overall item similarity (e.g., recommending Alien for liking Blade Runner), offering a more context-aware approach suitable for content-based systems.

Reference:

- Le, Q., & Mikolov, T. (2014). Distributed Representations of Sentences and Documents. International Conference on Machine Learning (ICML). `https://arxiv.org/abs/1405.4053`

How SGNS Works — A Simplified Example

> Throughout the book, we won't be able to provide such detailed and simplified explanations for every algorithm. However, to move from simple to complex concepts without overwhelming the reader, we'll explain the training and loss mechanisms using a practical example. The training process appears in many algorithms, so it's best to understand it well early on, while the algorithms are still relatively simple.
>
> For those who already understand how model training works, feel free to skip this chapter.

Imagine we have a dataset of 20 movies, each labeled with genres. The algorithm's objective is to learn a representation (a "map") where genres with similar meanings end up close together, while dissimilar genres are positioned far apart.

Collecting Positive Pairs

First, we process the movie data to identify pairs of genres that appear *together* in the same movie. For instance, if a movie is tagged as follows:

- **Movie:** Avatar
- **Genres:** `[Action, Adventure, Fantasy]`

We can derive several "positive pairs" from this single movie, representing genres that co-occur: `(Action, Adventure)`, `(Action, Fantasy)`, `(Adventure, Action)`, `(Adventure, Fantasy)`, and so on. These pairs serve as the "correct answers" or ground truth examples for the model. In the code provided in the tutorial, these are stored in the `positive_pairs` list.

(Note: In this simplified movie genre example, we treat pairs as undirected co-occurrences for clarity. In the original Word2Vec context using text, pairs are directional; the center word predicts context words within a defined window, without necessarily reversing the relationship automatically.)

The Challenge and the Negative Sampling Solution

The initial Skip-gram idea was conceptually straightforward: given the vector for a 'center' genre like 'Action', try to predict the vector for a co-occurring 'context' genre like 'Adventure'.

However, this presents a significant computational challenge. To predict 'Adventure', the model would need to calculate a relevance score for 'Adventure' compared to *every other genre* in the vocabulary (e.g., 'Romance', 'Sci-Fi', 'Comedy', etc.) and determine that 'Adventure' is the best fit. This becomes extremely slow and resource-intensive, especially with large vocabularies (imagine hundreds of thousands of words instead of just 20 genres).

Negative Sampling offers an elegant solution by simplifying the task. Instead of asking the model to "pick the right answer from the entire vocabulary," it reframes the problem as a series of much simpler binary ("Yes/No") classification questions.

A Single Training Step Explained

During each step of the training process, the model performs the following operations:

First, it takes one positive pair, such as `(Action, Adventure)`, designating 'Action' as the **"Center"** feature and 'Adventure' as the **"Context"** feature.

Second, it selects a small number, `k`, of "negative samples" – genres that *do not* belong with the center feature in this context. For example, if k=5, it might randomly pick `Romance`, `Musical`, `Crime`, `Drama`, and `Comedy` from the overall genre vocabulary. These serve as the "wrong answers."

(In practice, negative sampling often uses a specific probability distribution, like the unigram frequency raised to the power of 0.75, to slightly favor less common genres, making the learning more efficient. However, simple random sampling provides a good conceptual approximation.)

Third, the model is presented with the positive pair and the negative pairs. Its task is to distinguish between them, aiming for the following target scores:

Pair	Is this a real pair?	Model's Score (Target)
`(Action, Adventure)`	**Yes** (Positive)	"I want this to be **1**"
`(Action, Romance)`	**No** (Negative)	"I want this to be **0**"
`(Action, Musical)`	**No** (Negative)	"I want this to be **0**"
`(Action, Crime)`	**No** (Negative)	"I want this to be **0**"
`(Action, Drama)`	**No** (Negative)	"I want this to be **0**"
`(Action, Comedy)`	**No** (Negative)	"I want this to be **0**"

Before training begins, the system initializes an embedding table (represented as `nn.Embedding` in the code). This table contains one row for each unique genre, and each row is a vector of a predefined dimension (e.g., `embed_dim = 100`). Initially, these vectors are filled with random numbers. The training process uses the target scores (1s and 0s) as an "answer key" to iteratively update these random vectors, gradually making them meaningful representations of the genres.

Now, the 0/1 table comes into play. We use it as the "answer key" to update these random numbers and make them meaningful. The next section explains this update process.

Learning from Mistakes How Vectors Adjust

The model starts with no knowledge and makes predictions based on its current random vectors. It uses the **Dot Product** to estimate the similarity between the center genre's vector and the context/negative genre's vector.

The dot product provides a mathematical measure of vector similarity:

- If two vectors point in similar directions, their dot product is a large positive number.
- If they are unrelated (orthogonal), the dot product is near zero.
- If they point in opposite directions, the dot product is a large negative number.

This prediction calculation is often referred to as the "forward pass" in neural network terminology. For a positive pair like `(Action, Adventure)` and a negative sample like `(Action, Romance)`, the process involves looking up the current vectors (e.g., `Action`: `[0.1, 0.5]`, `Adventure`: `[0.2, 0.4]`, `Romance`: `[-0.3, 0.1]`) and computing their dot products:

- Positive Score: `(0.1 * 0.2) + (0.5 * 0.4) = 0.22`
- Negative Score: `(0.1 * -0.3) + (0.5 * 0.1) = 0.02`

These raw scores (`0.22`, `0.02`) represent the model's initial similarity guesses. They are then passed through the **Sigmoid function**, which squashes any number into a probability between 0 and 1.

- `sigmoid(0.22)` might yield `0.55` (Model's guess: "55% likely to be a real pair").
- `sigmoid(0.02)` might yield `0.51` (Model's guess: "51% likely to be a real pair").

Now, the learning begins by calculating the error or **Loss**. The model compares its predicted probabilities to the target scores (1 for positive, 0 for negative). The loss quantifies how far off the predictions are. For a positive pair, the error is calculated using `-log(sigmoid(positive_score))` ; we want this low, meaning the predicted probability should be close to 1. For negative pairs, the error is `-log(sigmoid(-negative_score))` ; we want this low, meaning the predicted probability should be close to 0. (The formula `log(sigmoid(-neg_dot))` seen in code is a numerically stable way to compute `log(1 - sigmoid(neg_dot))` .) The total loss for the training step is typically the sum or average of these individual errors.

Next, the system calculates **Gradients** using backpropagation (triggered by `loss.backward()` in PyTorch). A gradient indicates the direction and magnitude of change needed for each number in the embedding vectors to reduce the total loss. It effectively tells each vector how to adjust:

- The 'Action' vector might be told to move slightly to increase its similarity score with 'Adventure' and decrease its score with 'Romance'.
- The 'Adventure' vector is nudged closer to 'Action'.
- The 'Romance' vector is pushed further away from 'Action'.

Finally, the **Optimizer** (e.g., Adam) uses these gradients to **update** the vectors in the embedding tables (`optimizer.step()` in PyTorch). This step actually modifies the numerical values based on the calculated adjustments.

This entire process—predict, calculate loss, compute gradients, update vectors—is repeated thousands of times across all positive pairs in the dataset. Through this constant adjustment, vectors for genres that frequently co-occur (like 'Action', 'Adventure', 'Sci-Fi') are pulled closer together , while vectors for dissimilar genres are pushed apart. This iterative "tug-of-war" ultimately organizes the vectors into a meaningful "map" where proximity reflects semantic similarity.

The training typically runs for multiple epochs (full passes over the data), using higher-dimensional vectors (e.g., 100-300 dimensions) than the simple 2D example shown here. Hyperparameters like the number of negative samples (`k`), the learning rate, and embedding dimension significantly influence the final quality of the learned representations.

Now let's look at the code.

Tutorial #1: Content-Based Similarity with Feature Embeddings

The tutorial sections in this guide are entirely optional, though they can be extremely valuable for building intuition and a deeper understanding of the algorithms in practice. They assume, however, a basic familiarity with the libraries and frameworks being used. If you are interested in exploring why a particular method, function, or framework class is invoked at a specific point in the code, we encourage you to consult the Appendix, where we provide detailed, line-by-line commentary for selected algorithms from the Tutorial sections.

This tutorial implements the content-based embedding model as described in the theoretical section. Instead of using user interaction data, we will learn embeddings for **item features** (movie genres) to find items with similar content.

The core idea is:

(1) Treat each movie's list of genres as a "sentence."

(2) Use a Skip-gram model to learn dense vector embeddings for each genre.

(3) Represent each movie by averaging the embeddings of its genres.

(4) Recommend movies by finding those with the highest Cosine Similarity.

For the line-by-line explanations of the code samples, please see the Appendix.

Step 1: Prepare Data from Movie Genres

First, we load the `u.item` file from the MovieLens 100k dataset. This file contains movie titles and their associated genres, which will serve as our content features.

```python
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import pandas as pd
from torch.utils.data import Dataset, DataLoader
from collections import defaultdict

#  Data Loading and Preparation

movie_df = pd.read_csv(
    'ml-100k/u.item', sep='|', encoding='latin-1', header=None,
    names=['movie_id', 'title'] + list(range(22))
)

# Extract genres for each movie
genre_cols = movie_df.columns[-19:]
movie_genres = {}
for index, row in movie_df.iterrows():
    genres = [
        genre_cols[i] for i, val in enumerate(row[genre_cols]) if val
    ]
    if genres: # Only include movies with at least one genre
        movie_genres[row['movie_id']] = genres

#  Vocabulary and Pair Generation

all_genres = sorted(list(set(
    g for genres in movie_genres.values() for g in genres
)))
genre_to_idx = {genre: i for i, genre in enumerate(all_genres)}
vocab_size = len(all_genres)
print(f"Found {len(movie_genres)} movies and {vocab_size} genres.")

# Generate positive pairs from co-occurring genres in the same movie
positive_pairs = []
for genres in movie_genres.values():
    if len(genres) >= 2:
        indices = [genre_to_idx[g] for g in genres]
        for i in range(len(indices)):
            for j in range(len(indices)):
                if i != j:
                    positive_pairs.append((indices[i], indices[j]))
print(f"Generated {len(positive_pairs)} positive genre pairs.")

# Create negative sampling distribution from genre frequency
genre_freq = defaultdict(int)
for genres in movie_genres.values():
    for g in genres: genre_freq[g] += 1

freq_list = [genre_freq[g] for g in all_genres]
prob = np.array(freq_list) ** 0.75
prob /= prob.sum()
prob_tensor = torch.from_numpy(prob).float()
```

```
# Setup DataLoader for training
class GenrePairsDataset(Dataset):
    def __len__(self): return len(positive_pairs)
    def __getitem__(self, idx): return torch.tensor(positive_pairs[idx])

dataloader = DataLoader(GenrePairsDataset(), batch_size=1024, shuffle=True)
```

Step 2: Define the SkipGramNS Model

The model architecture uses PyTorch's `nn.Embedding` layers to store the genre vectors. Its `forward` method implements the Skip-gram with Negative Sampling loss function, which trains the model to give high scores to co-occurring genres and low scores to random pairs.

```
class SkipGramNS(nn.Module):
    def __init__(self, vocab_size, embed_dim):
        super().__init__()
        self.in_embed = nn.Embedding(vocab_size, embed_dim)
        self.out_embed = nn.Embedding(vocab_size, embed_dim)
        self.in_embed.weight.data.uniform_(-1, 1) # Init weights
        self.out_embed.weight.data.uniform_(-1, 1)

    def forward(self, centers, contexts, negatives):
        v_c = self.in_embed(centers)
        v_o = self.out_embed(contexts)
        pos_dot = (v_c * v_o).sum(1)
        pos_loss = torch.log(torch.sigmoid(pos_dot))

        v_neg = self.out_embed(negatives)
        v_c_unsq = v_c.unsqueeze(2)
        neg_dot = torch.bmm(v_neg, v_c_unsq).squeeze(2)
        neg_loss = torch.log(torch.sigmoid(-neg_dot)).sum(1)

        loss = - (pos_loss + neg_loss).mean()
        return loss
```

Step 3: Train the Model to Learn Genre Embeddings

The training loop feeds batches of positive genre pairs to the model. For each pair, it also draws `k` negative samples. The optimizer then adjusts the embedding weights to minimize the loss, gradually teaching the model the relationships between genres.

To understand why we use Adam: Recall the mathematical objective from the Skip-gram with Negative Sampling (NS). The objective function for each training example is:

$$\mathcal{J} = \log \sigma(v'_{f_o} \cdot v_{f_c}) + \sum_{i=1}^{k} \mathbb{E}_{f_i \sim P_n(f)} [\log \sigma(-v'_{f_i} \cdot v_{f_c})]$$

In practice, we minimize the loss $\mathcal{L} = -\mathcal{J}$ using gradient descent. Adam is an adaptive variant of Stochastic Gradient Descent (SGD) that adjusts the learning rate for each parameter based on the history of gradients, making it more efficient and stable for training embedding models like this one. It helps converge faster than plain SGD, especially with noisy data from negative sampling, by incorporating momentum and adaptive scaling (as described in the Adam paper by Kingma & Ba, 2014).

For negative sampling, we use `torch.multinomial` to draw k negative samples per positive pair from the precomputed probability distribution (`prob_tensor`). This distribution is the unigram frequency raised to the 0.75 power $P_n(f) \propto \text{freq}(f)^{0.75}$, as recommended in the original Word2Vec paper (Mikolov et al., 2013). `torch.multinomial` efficiently samples from this categorical distribution without replacement, simulating the expectation $\mathbb{E}_{f_i \sim P_n(f)}$ in the objective function. For very large vocabularies (e.g., thousands of unique items), this can be computationally expensive due to repeated sampling; optimizations like subsampling frequent items (to reduce noise from common words/movies) or hierarchical softmax (as an alternative to NS) may improve efficiency in such cases.

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = SkipGramNS(vocab_size, embed_dim=30).to(device)
optimizer = optim.Adam(model.parameters(), lr=0.01)
k, epochs = 5, 20

print("\nStarting training for genre embeddings...")
for epoch in range(epochs):
    total_loss = 0
    for batch in dataloader:
        centers, contexts = batch[:, 0].to(device), batch[:, 1].to(device)
        # Draw k negative samples for each positive pair
        negs = torch.multinomial(prob_tensor, len(centers) * k, True)
        negs = negs.view(len(centers), k).to(device)

        loss = model(centers, contexts, negs)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    print(f"Epoch {epoch+1}: Loss = {total_loss/len(dataloader):.4f}")

# Extract the learned genre embeddings for use in recommendation
genre_embeddings = model.in_embed.weight.data.cpu().numpy()
```

Step 4: Generate Content-Based Recommendations

With the genre embeddings learned, we can now build our recommender. We represent each movie by averaging the embedding vectors of its genres. Recommendations are then generated by calculating the cosine similarity between these aggregated movie vectors.

```python
# Create a vector for each movie by averaging its genre embeddings
movie_vectors = {}
for movie_id, genres in movie_genres.items():
    if genres:
        genre_indices = [genre_to_idx[g] for g in genres]
        movie_vectors[movie_id] = genre_embeddings[genre_indices].mean(0)

# Convert to a matrix and list for efficient similarity calculation
movie_ids_list = list(movie_vectors.keys())
movie_matrix = np.array([movie_vectors[mid] for mid in movie_ids_list])

def get_similar_movies(target_title, top_n=5):
    target_row = movie_df[movie_df['title'] == target_title]
    if target_row.empty:
        print(f"Movie '{target_title}' not found.")
        return

    target_id = target_row['movie_id'].iloc[0]
    target_idx = movie_ids_list.index(target_id)
    target_vec = movie_matrix[target_idx]

    # Calculate cosine similarity between the target and all others
    sims = (movie_matrix @ target_vec) / (
        np.linalg.norm(movie_matrix, axis=1) * np.linalg.norm(target_vec)
    )

    # Get top N indices, excluding the first one (the item itself)
    sim_indices = np.argsort(sims)[::-1][1:top_n+1]

    print(f"\nMovies similar to '{target_title}':")
    for idx in sim_indices:
        sim_id = movie_ids_list[idx]
        sim_title = movie_df[movie_df['movie_id']==sim_id]['title'].iloc[0]
        score = sims[idx]
        print(f"- {sim_title}: Sim = {score:.4f}")

get_similar_movies('Toy Story (1995)', top_n=5)
get_similar_movies('Star Wars (1977)', top_n=5)
```

Output:

```
 Step 1: Preparing Data

Found 1682 movies with genres and 19 unique genres.
Generated 3350 positive genre pairs.
 Step 2: Training Model

Using device: cpu
Epoch 5/20: Avg Loss = 3.3323
Epoch 10/20: Avg Loss = 2.5826
Epoch 15/20: Avg Loss = 2.4474
Epoch 20/20: Avg Loss = 2.3943
 Step 3: Generating Recommendations

Movies similar to 'Toy Story (1995)' (Genres: [6, 7, 8]):
- Aladdin and the King of Thieves (1996) (Genres: [6, 7, 8]): Sim = 1.0000
- Goofy Movie, A (1995) (Genres: [6, 7, 8, 17]): Sim = 0.9559
- Aladdin (1992) (Genres: [6, 7, 8, 15]): Sim = 0.9441
- Little Rascals, The (1994) (Genres: [7, 8]): Sim = 0.8897
- Mouse Hunt (1997) (Genres: [7, 8]): Sim = 0.8897

Movies similar to 'Star Wars (1977)' (Genres: [4, 5, 17, 18, 20]):
- Return of the Jedi (1983) (Genres: [4, 5, 17, 18, 20]): Sim = 1.0000
- Starship Troopers (1997) (Genres: [4, 5, 18, 20]): Sim = 0.9505
- Empire Strikes Back, The (1980) (Genres: [4, 5, 11, 17, 18, 20]): Sim = 0.9484
- African Queen, The (1951) (Genres: [4, 5, 17, 20]): Sim = 0.9408
- Last of the Mohicans, The (1992) (Genres: [4, 17, 20]): Sim = 0.9117

Movies similar to 'Scream (1996)' (Genres: [14, 19]):
- Village of the Damned (1995) (Genres: [14, 19]): Sim = 1.0000
- Scream (1996) (Genres: [14, 19]): Sim = 1.0000
- Believers, The (1987) (Genres: [14, 19]): Sim = 1.0000
- Nightwatch (1997) (Genres: [14, 19]): Sim = 1.0000
- Loch Ness (1995) (Genres: [14, 19]): Sim = 1.0000
```

Note on Similarity Scores of 1.0:

As seen in the recommendations for 'Scream (1996)', several movies have a perfect similarity score of 1.0. This is not an error but a direct consequence of our methodology. Since we represent each movie solely by averaging its genre embeddings, any two movies that share the exact same set of genres will have identical vector representations. The cosine similarity between identical vectors is always 1.0. This highlights a limitation of using a small, discrete set of features; to achieve more nuanced recommendations, one could incorporate additional content features like keywords, tags, or director/actor information.

1.1.4 Limitations of Content-Based Filtering

- **Limited Serendipity.** Since it recommends items similar to what a user already likes, it can struggle to introduce novel and diverse content. This can lead to an over-specialization of recommendations.
- **Requires Rich Metadata.** The performance of content-based methods is entirely dependent on the quality and richness of the item features. Items with sparse or no metadata cannot be accurately represented and are difficult to recommend.
- **Inability to Leverage Collaborative Information.** These methods are "deaf" to the wisdom of the crowd. They cannot capture preferences that emerge from user-item interaction patterns, such as learning that users who like item A also tend to like item C, even if A and C have no features in common.

Production Ready?

- **Content-Based Filtering: Absolutely Production-Ready.** These techniques are robust, scalable, and widely deployed. TF-IDF is a cornerstone of information retrieval and serves as a powerful baseline in many production systems. Semantic models using Word2Vec/Doc2Vec are also common for enriching item representations.

TF-IDF, Cosine Similarity:

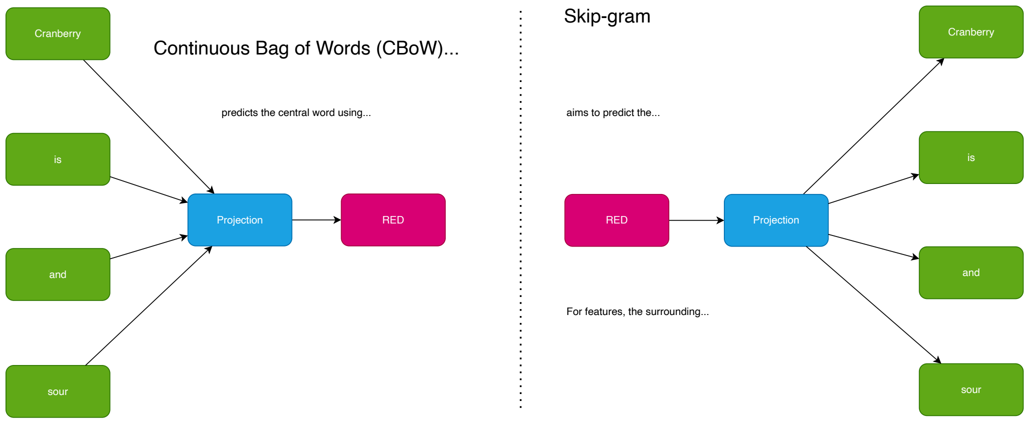
- **scikit-learn** : The definitive library for these tasks. It provides highly optimized and easy-to-use implementations of `TfidfVectorizer` and `cosine_similarity`.

Word2Vec, Doc2Vec:

- **Gensim** : The standard, go-to library for training and using Word2Vec and Doc2Vec models.
- **spaCy** : A powerful NLP library that provides pre-trained word embeddings and easy access to document vectors.

1.1.5 Alternative Word2Vec Architecture: CBOW

As an alternative to the Skip-gram model, the **Continuous Bag-of-Words (CBOW)** architecture offers a different but related approach to learning feature embeddings. While Skip-gram predicts context features from a central feature, CBOW does the reverse: it predicts the central feature based on the sum or average of its surrounding context feature embeddings.


Cranberry
Continuous Bag of Words (CBoW)...
predicts the central word using...
is
Projection
RED
and
sour
Skip-gram
aims to predict the...
Cranberry
is
RED
Projection
and
For features, the surrounding...
sour

Key concept — CBOW is more computationally efficient and tends to perform better for frequent features. For a given set of context features (e.g., `['cyberpunk', 'dystopian']` for a movie), the model is trained to predict the most likely center feature (`'sci-fi'`). This approach smooths over the distributional information from the entire context, making it faster to train.

However, this averaging of context features, which makes CBOW efficient, can also be a limitation in certain recommendation scenarios. By blending the context features into a single average vector, CBOW can obscure the influence of specific individual features or ignore the sequential order in which interactions occurred. For example, if a user interacts with `[ItemA (Sci-Fi), ItemB (Comedy), ItemC (Sci-Fi)]`, CBOW tries to predict `ItemB` from the average of `ItemA`'s and `ItemC`'s vectors (which might represent Sci-Fi). This averaging makes it difficult for the model to learn the specific association between `ItemB` and its distinct neighbors if the average context points in a different semantic direction.

Furthermore, if the context contains a mix of very common and very rare features, the average vector might be dominated by the frequent features, potentially diluting the signal from the rare but important ones. This can make it harder for CBOW to accurately predict less common central features or learn nuanced relationships, especially when user intent is driven by specific item sequences rather than a general thematic average. In contrast, Skip-gram, which predicts context features *from* a central feature, focuses on capturing pairwise relationships more directly and might be more effective in scenarios where specific feature co-occurrences or sequence order are critical signals.

When to Consider — Use CBOW when training speed is a priority and your dataset has a sufficient number of examples for your key features. It serves as an efficient alternative to Skip-gram for learning dense feature representations, but evaluate its suitability based on whether the assumption of an ”average context” adequately captures the user behavior patterns relevant to your recommendation task.

1.1.6 Handling Rare and Unknown Features: Subword Embeddings (FastText)

A significant limitation of standard Word2Vec models (both Skip-gram and CBOW) is their inability to handle out-of-vocabulary (OOV) features—items that were not seen during training. The **FastText** model solves this by learning embeddings for **character n-grams**, rather than entire features.

Key concept — FastText represents each feature as a bag of its constituent character n-grams. For example, the feature `sci-fi` might be broken down into n-grams like `<sc`, `sci`, `ci-`, `i-f`, `-fi`, `fi>` (where `<` and `>` denote the start and end of the feature). The model learns embeddings for these n-grams, and the final vector for a feature is the sum of its n-gram vectors.

> **It should be noted that this algorithm is more applicable to text processing than to objects with features, which are the basis of content-based methods. Therefore, it is presented here mainly for completeness and consistency.**

Key differentiator — This subword approach allows the model to construct a meaningful vector for a previously unseen feature by combining the vectors of its parts. For instance, if the model has seen ”sci-fi” and ”romance,” it can generate a reasonable vector for the OOV feature ”sci-mance” because it recognizes the component n-grams.

Use cases — This is extremely valuable for recommendation systems dealing with dynamic or niche catalogs where new item attributes (features) appear frequently. It provides a robust solution to the **feature cold-start problem** and generally improves the quality of embeddings for rare features.

1.2 Rule-Based and Heuristic Systems

Rule-based and heuristic systems represent the most transparent and direct approach to recommendation. They operate not on learned user-interaction patterns but on manually defined logic or simple, universal metrics. These methods are highly interpretable, easy to implement, and serve as crucial components for handling business constraints, promotions, and the cold-start problem where interaction data is absent.

However, this reliance on manually crafted logic is also their primary weakness. Such systems lack the ability to learn from data or adapt to changing user behavior automatically. As the number of rules and contextual factors grows, the system can become brittle and cumbersome to maintain, with conflicts between rules requiring complex, manual resolution. Furthermore, because they operate on explicit instructions rather than discovered patterns, they struggle to provide truly personalized or serendipitous recommendations, limiting their ability to uncover novel user interests that are not already encoded in the business logic.

1.2.1 Classic Rule-Based Systems

Key concept — Classic rule-based systems use manually defined, domain-specific **logical predicates** (often in an "if-then" structure) to generate recommendations. These rules are crafted based on business knowledge and do not involve statistical learning from user data; they are a direct implementation of business logic.

Key differentiator — Their complete transparency and control. The reason for a recommendation is never a black box; it's a direct consequence of a predefined rule (e.g., `IF user.location == "New York" AND context.season == "winter" THEN recommend item.category == "heavy coats"`). This makes them predictable, easy to debug, and simple to modify based on changing business needs.

Drawback — Their primary limitation is scalability in terms of maintenance. As the number of rules grows, managing them, resolving conflicts, and updating them becomes increasingly complex and brittle.

Use cases — Implementing business-critical promotions ("Top 10 Bestsellers in Your Region"), strategic recommendations (promoting high-margin items), or providing sensible defaults for new users (e.g., showing the most popular items on the homepage). They are often used as a foundational layer or a filtering mechanism in a hybrid system.

When to Consider — Use a rule-based system when recommendations need to be 100% transparent, controllable, and aligned with explicit business goals. They are also invaluable for handling the "coldest start" scenario (a brand new site with no users or interactions) and for creating non-personalized but useful features like "Most Popular" or "New Arrivals" lists.

1.2.2 Heuristic-Driven Algorithms

These algorithms are not personalized to individual users but instead rely on global aggregate metrics. They are simple, computationally efficient, and serve as powerful baselines.

1.2.2.1 Top Popular (Most Popular)

Key concept — This heuristic recommends items based on their overall popularity across all users. Popularity is typically measured by the total number of interactions (e.g., views, purchases, or ratings).

Mathematical Foundations

Let I be the set of all items in the catalog. For each item i in that set, we define a popularity score $P(i)$ as the total count of users who have interacted with it.

To calculate this, let U be the set of all users. The score $P(i)$ is the size of the set of users u who have interacted with item i. This is written formally as:

$$P(i) = |\{u \in U \mid \text{user } u \text{ interacted with item } i\}|$$

Here, the vertical bars $|\cdot|$ mean "the size of the set." So, the formula simply counts every user who has a record of interacting with that specific item. The recommendation list is then generated by sorting all items in descending order of their score $P(i)$ and taking the top N.

Use cases — It is the most common non-personalized recommendation strategy, often used as a fallback for new users or in "Top 10" lists on homepages. It effectively mitigates the cold-start problem by providing a sensible default.

1.2.3 Association Rule Mining

Association Rule Mining discovers "if-then" patterns from transactional data. In the context of recommendation, this translates to finding rules like {`item A`} `->` {`item B`}, which means that users who interacted with item A are also likely to interact with item B. This is the fundamental logic behind ubiquitous e-commerce features like "Frequently Bought Together" and "Customers Who Viewed This Also Viewed".

The "if" part of the rule ({`item A`}) is known as the **antecedent**, while the "then" part ({`item B`}) is the **consequent** . While simple rules involving single items are common, the real challenge arises when dealing with thousands of items, leading to a potential explosion in the number of possible rules (e.g., {`A, B`} `->` {`C`}, {`A, B, C`} `->` {`D`}). To manage this complexity, ARM relies on specific metrics and efficient algorithms, explained in this chapter.

Unlike collaborative filtering, which models user profiles, this method is item-centric and operates on a simple principle of co-occurrence. The process typically involves two major steps:

(1) **Frequent Itemset Mining:** First, identify all sets of items that frequently appear together in transactions. (2) **Rule Generation:** Second, generate reliable "if-then" rules from these frequent itemsets.

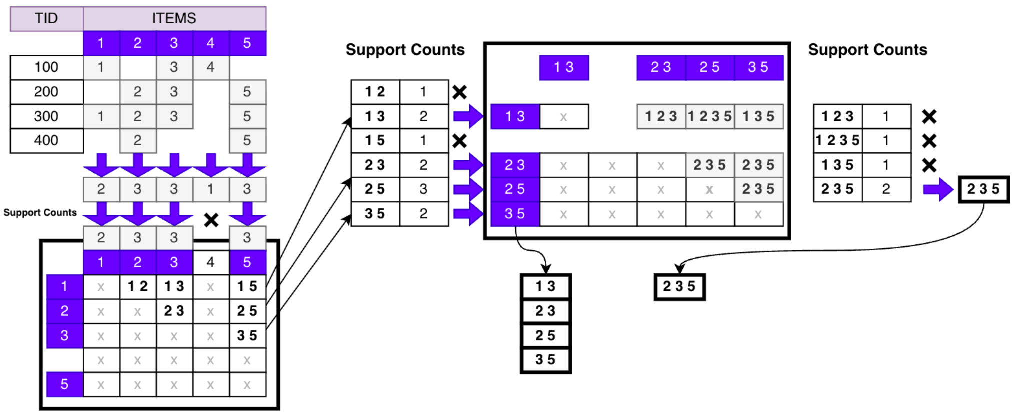
1.2.3.1 Frequent Itemset Mining Algorithms

The computational core of association rule mining is finding the frequent itemsets. Several algorithms have been developed for this task, each with different strategies and trade-offs.

1.2.3.1.1 Apriori: The Classic Approach The Apriori algorithm is a foundational method for frequent itemset mining, efficiently identifying all itemsets in a dataset that satisfy a minimum support threshold (`min_support`). It leverages the **Apriori Principle**: *if an itemset is frequent, then all of its subsets must also be frequent.* This principle allows the algorithm to prune a large number of candidate itemsets, making the search computationally feasible. Apriori works iteratively, first finding frequent individual items, then frequent pairs, triplets, and so on, pruning candidates at each step. Its primary bottleneck is the computationally expensive process of generating and testing a vast number of candidate itemsets, particularly for datasets with many unique items.

Mathematical Foundations & Walkthrough

Apriori uses a level-wise, breadth-first search to find frequent itemsets. It operates in passes, where pass k identifies all frequent itemsets of size k, denoted L_k.



Apriori works **level by level**, finding frequent itemsets of increasing size (first size 1, then size 2, then size 3, and so on). Each pass (or level k) involves three main stages:

1. Generate Potential Candidates (Join Step)

The goal of this step is to create a list of *potential* frequent itemsets for the current size, k. The algorithm uses the list of *actual* frequent itemsets found in the *previous* level (L_{k-1}) and joins pairs of these smaller sets to make candidate sets of size k.

The rule for combining is precise: It joins two frequent $k-1$ itemsets only if they are identical except for their last item (assuming the items in the sets are sorted). For example, if {`Milk, Bread`} (from L_2) and {`Milk, Butter`} (from L_2) were both frequent, the algorithm joins them to create the candidate {`Milk, Bread, Butter`} (a candidate for L_3).

2. Prune Candidates (Apriori Principle)

This step filters out candidates that *cannot* be frequent, saving significant computation.

Before even looking at the transaction data, the algorithm applies a crucial check based on the **Apriori Principle**: For any candidate itemset (size k), it examines all its smaller subsets (size $k-1$). If *any* of these smaller subsets were *not* on the frequent list from the previous level (L_{k-1}), then the candidate itself cannot be frequent and is discarded.

For example, if {`Milk, Bread, Butter`} is a size-3 candidate, the algorithm checks its size-2 subsets: {`Milk, Bread`}, {`Milk, Butter`}, and {`Bread, Butter`}. If it turns out that {`Bread, Butter`} was *not* on the list of frequent 2-itemsets (L_2), the algorithm immediately discards {`Milk, Bread, Butter`} without needing to count it in the database.

3. Filter Candidates (Count Support)

For the candidates that survive the pruning step, the algorithm finally scans the database. It counts how many transactions contain each surviving candidate itemset.

The algorithm keeps only those candidates whose count (support) is greater than or equal to the predefined `min_support` threshold. These surviving candidates become the *actual* frequent itemsets for that level (L_k).

Apriori repeats these "Generate," "Prune," and "Filter" stages, increasing the itemset size k by one each time (size 1 $\rightarrow$ size 2 $\rightarrow$ size 3...). The process stops when a level is reached where no new frequent itemsets are found, as the Apriori principle guarantees that no larger sets can be frequent.

Example

To illustrate Apriori, consider a grocery store dataset with 5 transactions and a `min_support` of 2 transactions (i.e., itemsets must appear in at least 40% of transactions to be frequent).

Original Transactions:

- T1: {Milk, Bread, Butter}
- T2: {Milk, Bread}
- T3: {Milk, Butter}
- T4: {Bread, Butter}
- T5: {Milk}

Iteration 1 (k=1): Find Frequent 1-Itemsets

- **Scan Database**: We first find the **support count** for each item. (We'll introduce the formal, proportional definition of 'support' along with confidence and lift later. For now, 'Support' refers to the absolute number of transactions an item appears in.)
  - SupportCount({Milk}) = 4
  - SupportCount({Bread}) = 3
  - SupportCount({Butter}) = 3
- **Prune**: All items have supportCount `>=` min_support which is `2`.
- **Result** (L_1): $L_1 = \{ \{\text{Milk}\}, \{\text{Bread}\}, \{\text{Butter}\} \}$

Iteration 2 (k=2): Find Frequent 2-Itemsets

Join Step: Generate C_2 by joining L_1 with itself.

$C_2 = \{ \{\text{Milk, Bread}\}, \{\text{Milk, Butter}\}, \{\text{Bread, Butter}\} \}$

Prune Step: All 1-item subsets are in L_1, so no candidates are pruned yet. Scan the database:

- SupportCount({Milk, Bread}) = 2 (in T1, T2)
- SupportCount({Milk, Butter}) = 2 (in T1, T3)
- SupportCount({Bread, Butter}) = 2 (in T1, T4)