SAPIENZA
UNIVERSITÀ DI ROMA

Demystifying Sequential Recommendations: Counterfactual Explanations via Genetic Algorithms and Automata Learning

Facoltà di Ingegneria dell'informazione, informatica e statistica
Master's Degree in Computer Science

Domiziano Scarcelli
ID number 1872664

Advisor
Prof. Gabriele Tolomei

Co-Advisor
Prof. Giuseppe Perelli

Academic Year 2024/2025

Demystifying Sequential Recommendations: Counterfactual Explanations via Genetic Algorithms and Automata Learning

Sapienza University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: scarcelli.1872664@studenti.uniroma1.it

Abstract

While Sequential Recommender Systems (SRSs) have demonstrated remarkable effectiveness in capturing evolving user preferences, their inherent complexity as "black box" models poses significant challenges for interpretability. This research proposes novel techniques for counterfactual explanations in SRSs, addressing the fundamental question of what minimal changes in a user's interaction history would lead to different recommendations. We introduce two distinct approaches: a specialized genetic algorithm optimized for discrete sequences, and an automata learning method that constructs an interpretable surrogate model. We evaluate these approaches across four experimental settings, varying between targeted/untargeted and categorized/uncategorized scenarios, to comprehensively assess their capability in generating meaningful explanations. Using MovieLens 100K and 1M datasets, we demonstrate that our methods successfully generate minimal, actionable counterfactuals that explain model decisions while maintaining high model fidelity. Our findings contribute to the growing field of Explainable AI by providing a framework for understanding sequential recommendation decisions through the lens of "what-if" scenarios, ultimately enhancing user trust and system transparency.

Contents

Chapter 1

Introduction

Personalized recommendations are at the core of modern digital experiences, shaping user interactions based on their interests, from movie and music streaming platforms, to e-commerce product suggestions, online advertising and social media content recommendation (Roy and Dutta, 2022).

Considering the huge amount of data present and uploaded daily on each of the aforementioned platforms, without personalized recommendation, the user would be lost. Recommender systems analyze this vast amount of user data, processing past interaction and preferences, merging it with contextual information to predict and suggest items that align with individual tastes and behaviors, highlighting more interesting and personalized content (Zhang et al., 2020), (Roy and Dutta, 2022), (Wachter et al., 2018a). This is crucial to maintain user engagement and satisfaction with the platform, but also to increase conversion rate and customer satisfaction by the business point of view. Let's just imagine YouTube, Netflix or Spotify without their personalized recommendations, the content will be hard to find.

Despite their widespread adoption and effectiveness, a major challenge remains: the lack of **interpretability**. Many state-of-the-art recommender systems, particularly those using deep learning and complex machine learning models, operate as **"black boxes"**—producing recommendations based on intricate, high-dimensional representations. In such systems, the output is the result of a complex, nonlinear function of the input, making it non-trivial to understand **why** a particular item was recommended over another. This opacity raises critical concerns for both users and system designers.

For end users, an unexplained recommendation can lead to skepticism and reduced trust in the system, particularly when recommendations appear unexpected or misaligned with their preferences. For businesses and system developers, the inability to interpret recommendation outcomes makes it difficult to diagnose errors, detect biases, and ensure fairness in decision-making (Longo et al., 2024). Moreover, as regulations such as the **General Data Protection Regulation (GDPR)** [1] emphasize the need for transparency

[1] Stated in Recital 71: https://gdpr-info.eu/recitals/no-71/

in automated decision-making, interpretability has become an essential requirement for real-world deployment (Wachter et al., 2018a).

Among the different types of recommender systems, **Sequential Recommender Systems (SRSs)** have gained increasing attention due to their ability to model user behavior over time. Unlike traditional recommender systems, which often assume that user preferences are static, SRSs recognize that user interests evolve dynamically based on recent interactions. By leveraging the sequential nature of user activities—such as browsing history, recent purchases, or watched content—these models capture temporal dependencies to generate more accurate and context-aware recommendations. This makes them particularly useful for applications where user preferences shift over time.

To address the challenges introduces by the lack of recommender systems' interpretability, in our work we will focus on SRSs explanations using **counterfactual examples** as a means to provide actionable insight into the model behavior. Counterfactual explanations aim to answer the question:

> What minimal changes in a user's interaction history would have led to a different recommendation?

By generating such explanations, we can offer users a clearer understanding of why a particular item was suggested and what factors influenced the model's decision.

Explainable AI (XAI)—the subfield of machine learning focusing on explaining opaque models—has received significant attention in recent years, with various methods developed to enhance the interpretability of machine learning models (Longo et al., 2024). However, fewer studies specifically address **counterfactual explanations**, particularly in **sequence-aware models**. This gap is even more pronounced when requiring counterfactuals for a specific target label (i.e., **targeted counterfactual explanations**), making interpretability in **sequential recommender systems (SRSs)** a relatively unexplored area.

Among general XAI techniques, **feature attribution methods** like **LIME** (Ribeiro et al., 2016) and **SHAP** (Lundberg and Lee) provide local interpretability by estimating the importance of input features. Meanwhile, **rule-based methods**, such as **LORE** (Guidotti et al., 2018), extract human-readable decision rules through decision tree learning. In the realm of **counterfactual explanations**, approaches such as **FACE** (Poyiadzi et al., 2020) ensure feasibility and actionability, while **MACE** (Karimi et al.) formulates counterfactual search as a Boolean satisfiability problem. Other methods, like **GeCo** (Schleich et al., 2021) and **CERTIFAI** (Sharma et al., 2020), leverage **genetic algorithms** to generate diverse and optimized counterfactuals, balancing similarity to the original input and predictive validity.

While most of these works focus on general machine learning models, some have been adapted for recommender systems. Notably, **ACCENT** (Tran et al., 2021) proposes action-based counterfactual explanations tailored for **neural recommenders**, providing tangible modifications to user behavior. Additionally, **CauseRec** (Zhang et al., 2021) applies counterfactual reasoning for data augmentation in **sequential recommendation**, improving user representations through contrastive learning.

Building on these prior works, we propose two novel counterfactual generation techniques

specifically designed for SRSs: **GENE** (**GEN**etic **C**ounterfactual **E**xplanations) and **PACE** (**P**ath-search **A**utomata **C**ounterfactual **E**xamples). The first leverages **a genetic algorithm** tailored for discrete sequences, optimizing counterfactual discovery based on sequence similarity and interpretability constraints. The second employs **automata learning**, constructing a surrogate model that enables both a **global view** of counterfactual states at each time step and **explicit counterfactual generation** for enhanced transparency. By integrating these methods, we aim to improve interpretability in SRSs while maintaining recommendation quality and user trust.

To comprehensively evaluate these approaches, we explore **four experimental settings**:

- **Untargeted-Uncategorized**: Generating counterfactual sequences with minimal changes to alter the recommendation, without any constraints on item categories.
- **Untargeted-Categorized**: Introducing item categories to assess how counterfactual explanations vary when considering higher-level semantic shifts.
- **Targeted-Uncategorized**: Ensuring that the counterfactual leads to a specific desired recommendation, increasing the challenge of solution sparsity.
- **Targeted-Categorized**: Combining category constraints with a target recommendation, providing structured and meaningful explanations.

We validate our methods on **MovieLens 100K and 1M datasets** using **model fidelity** as the evaluation metric, and **edit distance** as the distance metric, demonstrating their effectiveness in improving the interpretability of sequential recommendations.

The work is organized as follows: Chapter 2 introduces the concept of Explainable AI (XAI), explaining its importance and various explanation methods, including local surrogate models, feature attribution, rule-based explanations, and counterfactuals. It also highlights the key properties of effective counterfactual explanations.

Chapter 3 reviews related works in the field of Explainable AI, covering general methods such as LIME and SHAP, as well as those designed for recommender systems. It explores different types of explanations, including additive feature attribution, rule-based, and counterfactual explanations.

Chapter 4 introduces the four settings the method can handle: targeted vs. untargeted, and categorized vs. non-categorized. It then explores the techniques used to evaluate the validity of a potential counterfactual. The chapter details two proposed approaches: GENE, which is based purely on a genetic algorithm, and PACE, which builds on GENE and introduces an automaton to find counterfactuals using an ad-hoc path search algorithm. The chapter ends with a comparison of the two methods and a discussion of their computational complexity.

Chapter 5 explains the implementation details of the work, including the libraries used, evaluation strategies, hyperparameters, and train-test splitting methods.

Chapter 6 begins with an exploratory experiment that demonstrates how Sequence Recommendation Systems (SRSs) are sensitive to the last items in the sequence. It then presents

and discusses the evaluation results for both the GENE and PACE methods on two variants of the MovieLens dataset (100k and 1M) and three different models (GRU4Rec, SASRec, and BERT4Rec) across all four settings. The chapter also includes an evaluation of the accuracy of the learned automata.

Finally, Chapter 7 summarizes the work, discussing the limitations and potential future directions for the proposed methods.

Chapter 2

Background

2.1 Explainable AI

Explainable AI (XAI) (also referred to as **Interpretable AI**) is a subfield of Artificial Intelligence that focuses on developing methods to make AI model results interpretable to humans. The model or algorithm that explains the output of a system is called an *explainer* (or explainer model).

Over the years, XAI has grown significantly, evolving from a niche research topic into a widely studied and active field. This growth is driven by the increasing adoption of AI systems, particularly deep-learning-based techniques, which have demonstrated success across various domains, including finance, healthcare, and entertainment. However, the complexity of these models has made it crucial to understand their underlying mechanisms (Longo et al., 2024).

Deep learning models consist of an enormous number of parameters—often in the billions—each contributing to the final decision of the model. Due to this complexity, it is often challenging to determine a clear *cause-effect* relationship between input data and the model's output solely by analyzing them. This opacity has fueled the demand for explainability, ensuring that AI decisions are not only accurate but also interpretable and trustworthy.

2.1.1 Why do we need explanations?

The reasons on why explanations are useful are multiple. First, explanations enhance the user experience by helping users understand how decisions are made under the hood, rather than simply presenting the decision without any context. For instance, recommending a movie and giving an explanation of the type *"the movie X is recommended because you watched Y, and Z"* or *"movies Y, W, and Z are the movies that contributed more to the recommendation of X"* provide more value and clarity to the user compared to offering the recommendation alone.

Second, explanations may be crucial for ensuring regulatory compliance. For example, the

EU General Data Protection Regulation (GDPR) grants individuals the **right to an explanation** via requiring institutions to provide explanations to individuals that are subject to their (semi-)automated decision-making systems. (Karimi et al.).

Furthermore, explanations play a key role in building trust in automated systems, especially in high-stakes scenarios where incorrect decisions can lead to severe consequences. In areas such as autonomous driving or medical diagnostics, understanding the reasoning behind decisions is vital for ensuring safety and reliability, increasing confidence in the system's operation. For example, if the user is presented with a critical action to perform because the model suggested it, an explanation on why the model came up with that decision may be crucial for the human to better understand the reasoning, hence noticing possible errors. Explainability is in fact considered one of the four ethical principles for trustworthy AI, along with the respect for human autonomy, the prevention of harm and fairness [1].

Finally, explanations can also be useful during model development, as they help identify potential flaws or biases in AI models. This is crucial for enhancing model performance and ensuring the ethical use of AI systems. (Wachter et al., 2018b)

2.1.2 Explainers taxonomy

XAI explainers can be divided in different classes, depending on various factors. (Guidotti, 2024)

Explainability methods can be categorized based on the **level of access** they require to the underlying model:

- **White-box explainers** (also known as *model-specific explainers*) require full access to the model's internals, including weights, gradients, training data, and possibly source code. These methods are highly detailed, but may compromise privacy or proprietary information.
- **Gray-box explainers** operate with partial access, utilizing a subset of internal model information, such as gradients or training data, while still maintaining some level of abstraction.
- **Black-box explainers** (also called *model-agnostic explainers*) treat the model as an **oracle**, requiring no internal details. Instead, they rely solely on the model's input-output behavior to generate explanations, making them highly versatile and applicable across different models.

Black-box explainability is particularly valuable in scenarios where exposing internal model details is impractical or undesirable. Unlike white-box methods, which necessitate full transparency, black-box approaches allow data controllers to provide meaningful explanations while safeguarding sensitive user data and protecting proprietary trade secrets. This ensures a balance between transparency and confidentiality, making black-box explainers a practical choice for real-world applications. (Wachter et al., 2018b)

[1] As indicated in the "Ethics guidelines for trustworthy AI" at https://digital-strategy.ec.europa.eu/en/library/ethics-guidelines-trustworthy-ai

The second classification can be made depending on the **type of data** the explainer is capable of handling.

- **Data specific**: the explainer is designed to work with particular types of data, such as images, text, or tables.
- **Data agnostic**: the explainer can handle any type of data, regardless of its format.

The third and final classification depends on the **type of output** the explainer provides:

- **Endogenous explainers:** these provide examples or explanations based on the dataset itself. They select instances or use feature values directly from the given dataset.
- **Exogenous explainers:** these generate outputs that may not exist in the dataset. They rely on methods like interpolating between data points or using randomly generated data.

2.1.3 Types of explanations

An explanation is anything that makes a model's decision interpretable. Various types of explanations can be used to achieve this. Below, some of these types are described.

Local Surrogate Models

Deep learning models are opaque and usually not interpretable out-of-the-box—such as other linear models like Linear Regression or Decision Trees may be—mainly because of the high number of parameters; but their accuracy is usually higher than interpretable models. This phenomenon is known as **accuracy-interpretability tradeoff**, and it tells us that *the best explanation of a simple model is the model itself.* (Lundberg and Lee)

If the problem needs a higher accuracy model, a good way of explaining such opaque models is to create a simpler *explanation model* which is an interpretable approximation of the original model.

One widely used approach is the creation of **local surrogate models,** explored in the works by Ribeiro et al. (2016) and Guidotti et al. (2018). Local surrogate models are interpretable models, such as decision trees or linear regressions, that approximate the behavior of the complex model in a specific region of interest, typically around a particular instance.

Additive feature attribution

Additive feature attribution methods give an **importance score** to each feature in the input, which tells us which are the most influential features for the model final decision.

Formally, the explanation is a linear function of binary variables (Lundberg and Lee):

$$g\left(z'\right) = \phi_0 + \sum_{i=1}^{M} \phi_i z_i'$$

Here, $z' \in 0, 1^M$ represents the **simplified input features**, which are binary representations of the original input features. These simplified features indicate whether a specific feature is included ($z_i' = 1$) or excluded ($z_i' = 0$) in the explanation. The coefficients $\phi_i \in \mathbb{R}$ quantify the contribution of each feature to the model's output. The sum of these contributions, along with the baseline value ϕ_0, approximates the output $f(x)$ of the original, opaque model.

Two well-known **additive feature attribution methods** are **LIME** (Ribeiro et al., 2016) and **SHAP** (Lundberg and Lee):

- **LIME** generates explanations by creating a *local surrogate model* through perturbing the input x and querying the black-box model to observe how changes in the input affect predictions.
- **SHAP** generates feature importance scores by using Shapley values, a concept from cooperative game theory. These values are calculated by considering all possible combinations of features, measuring their contribution by adding or removing them from the model input. SHAP ensures consistency and fairness in the attribution of feature importance.

Rule-Based Explanations

Rule-based explanations aim to represent a model's decision-making process through easily interpretable **if-then rules.**

The **LORE** method (Guidotti et al., 2018) generates *rule-based explanations* by first building a **local surrogate model**, which in this case is a decision tree that approximates the behavior of the black-box model in the neighborhood of the input point being explained. By analyzing the structure of the decision tree, LORE derives **if-then rules** that describe the conditions under which the black-box model makes its predictions.

Rule-based explanations are particularly useful in domains like healthcare or finance, where interpretability and accountability are crucial.

Counterfactual Explanations

Introduced in the work by (Wachter et al., 2018b) as a method to provide an explanation without having to inspect the internals of the model, they take similar form to the statement:

> You were denied a loan because your annual income was £30,000. If your income had been £45,000, you would have been offered a loan. (Wachter et al., 2018b)

The explanation is formulated as the outcome a hypothetical set of actions different from the original set of actions would have produced. For a single source set of actions, multiple

counterfactual examples are possible. The explanation seek to find the counterfactual set of actions that is the closest as possible to the original set of actions.

Formally, given a classifier $\mathcal{F}$ that outputs a decision $y = \mathcal{F}(x)$ for an instance x, a **counterfactual explanation** consists of an instance $\tilde{x}$ such that the decision for $\mathcal{F}$ on $\tilde{x}$ is different from y (i.e., $\mathcal{F}(\tilde{x}) \neq \mathcal{F}(x)$).

In this work, we aim to tailor the counterfactual explanations to sequential recommender systems.

2.2 Recommender Systems

A recommender system (Figure 2.1) can be seen as a function $\mathcal{F}$ that, given a representation of the items a user u has interacted with, such as a one-hot vector $\mathbf{r}_u$—or a ratings vector—, it returns a vector of scores of the same length as the universe $\mathcal{I}$, which encodes the probability that the user u will interact with each item in the future.

$$\mathcal{F} : \mathbb{R}^{|\mathcal{I}_u|} \to \mathbb{R}^{|\mathcal{I}|}$$

Where $\mathcal{I}_u$ is the set of items the user has interacted with, represented as a one-hot vector, and $\mathcal{I}$ is the entire universe of items. The output of the function is a vector of relevance scores for all items in the universe.

Depending on the level of access to the system that implements the recommender system, the output may be more or less *transparent*. The raw scores are the most transparent output possible, but the system may return only the top-k scores or just the top-k items. (Ricci et al., 2011).

Example Suppose the universe is $\mathcal{I} = \{1, 2, 3, 4, 5\}$, and a user u has interacted with the items $\mathcal{I}_u = \{3, 6, 2, 5\}$, represented as a one-hot vector:

$$\mathbf{r}_u = [0, 1, 0, 0, 1, 1, 0]$$

The recommender system, $\mathcal{F}$, computes scores for each item, such as:

$$\mathcal{F}(\mathbf{r}_u) = [0.1, 0.1, 0.6, 0.05, 0.15]$$

Each value represents the predicted probability that the user will interact with each corresponding item in the universe. In this case, the recommender is providing full transparency by giving scores for all items.

The scores are often called **relevance scores**, and they reflect how strongly the system believes a user will engage with an item. These scores are derived from various factors, depending on the system type.

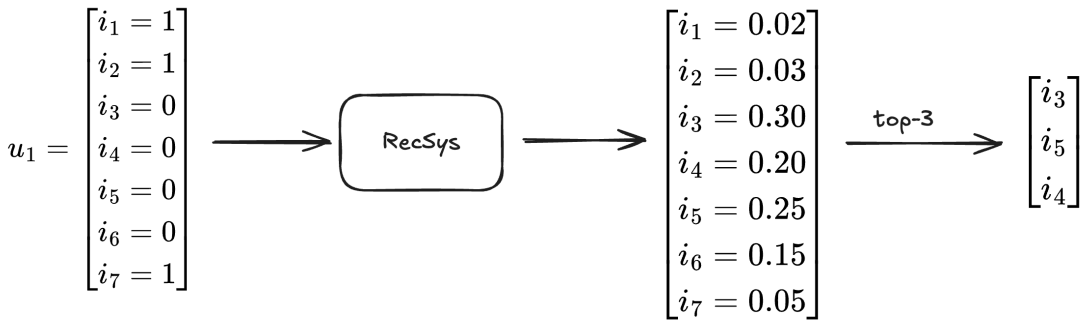


Figure 2.1. A diagram representing the recommender system input-output (with $k = 3$).

2.2.1 Content-Based Filtering

In **content-based filtering** (Figure 2.2), the recommender system computes the relevance score for an item based on the similarity between that item and the items a user has previously interacted with, denoted as $\mathcal{I}_u$. The similarity is usually derived from the item **features**, which can represent attributes like genre, tempo, author, publisher or other characteristics, depending on the domain (Roy and Dutta, 2022).

The basic intuition behind this approach is to recommend items whose features are similar to those of the items the user has already interacted with. In other words, content-based systems suggest items that align with items the user has interacted in the past based on their features.

In a music recommendation system, if a user has interacted with many songs that are characterized by *high tempo*, *rock*, and *electric guitar*, the system will recommend other songs that share similar features, such as those labeled with *rock*, *high tempo*, and *electric guitar*. The system does this by comparing the feature vectors of the items in the user's history with all other items in the universe.

While content-based filtering may be effective in many cases, it has some significant limitations:

- For each item in the universe $\mathcal{I}$, we need to compute a **feature vector** that represents the item's characteristics. Depending on the domain, this can be a challenging and non-trivial task. It may require domain-specific expertise and advanced techniques for extracting or defining relevant features (Roy and Dutta, 2022)
- To generate recommendations, the system must calculate the similarity between items in the user's history ($|\mathcal{I}_u|$ items) and all other items in the universe ($|\mathcal{I}|$ items). This results in a computationally expensive operation, as the system needs to perform $K \times M$ similarity comparisons, where $K = |\mathcal{I}_u|$ and $M = |\mathcal{I}|$. This becomes particularly costly when dealing with large datasets or long user histories.

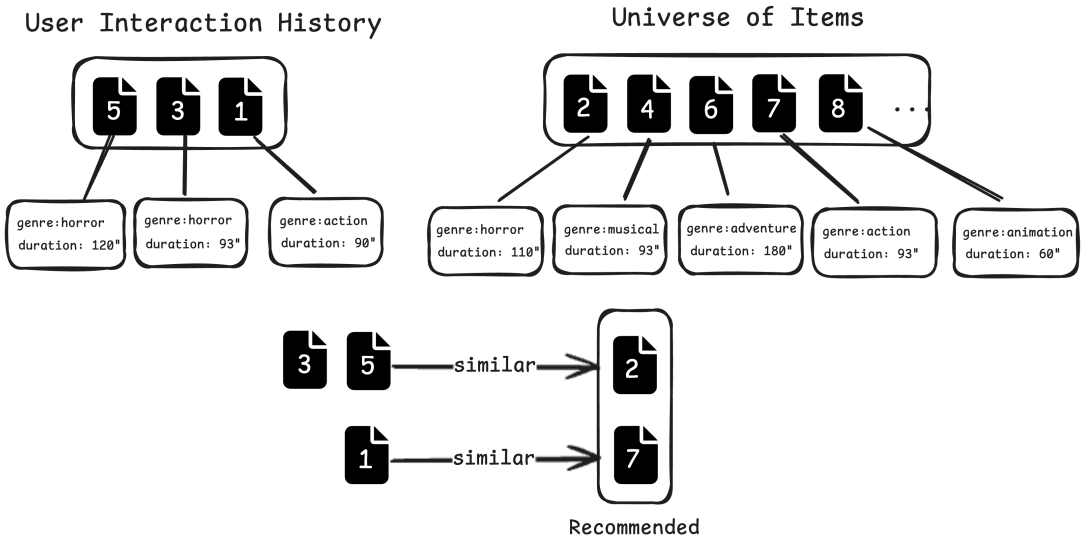


Figure 2.2. How content-based recommendation is performed: the features of each item in the user history are compared to the features of each other item. Items with similar features are recommended.

2.2.2 Collaborative Filtering

In **collaborative filtering** (CF), the relevance score for an item is determined by the similarity between a user's interaction history, $\mathcal{I}_u$, and the interaction histories of other users.

This approach overcomes some limitations of content-based filtering by relying on **user-item interaction patterns** rather than **explicit item features**.

The underlying assumption is that users with similar preferences will like similar items. The system will then identify those people and recommends items that similar users liked. (Breese et al., 2013), (Roy and Dutta, 2022).

Collaborative filtering can be implemented using two main approaches: **memory-based** and **model-based** techniques.

- **Memory-based** algorithms directly analyze the entire user-item interaction database, recommending new items by taking into consideration the preferences of its neighborhood (Roy and Dutta, 2022)
- **Model-based** algorithms aim to build a predictive model based on the user-item interactions, which is then used to predict the user's rating for unrated items, effectively providing recommendations (Breese et al., 2013), (Roy and Dutta, 2022).

In collaborative filtering, every user-item interaction provides **feedback**, indicating how much the user liked or interacted with an item. Feedback can be **explicit** or **implicit**.

An **explicit** feedback is a rating or preference the user explicitly provide to the system (as a rating on a scale of 1 to 5 stars); while an **implicit** feedback is something that is inferred from indirect indicators of user preferences, such as the number of views for a certain content, or the number of clicks on that particular product.

2.2.3 Collaborative Filtering Problems

Collaborative filtering methods suffer from several problems which can affect the quality of the recommendations. The most common issues are **cold start**, **scalability** and **sparsity of interactions**.

Cold start Cold start is a phenomenon caused by a lack of information on the user or item data. It can happen in three cases (Bobadilla et al., 2012):

1. **New community**: When the recommender is deployed for the first time, there are no existing users or interaction in the application. The system lacks the data required to make accurate recommendations, which will result in unreliable results.
2. **New item**: When a new item is added to the system and had never been an object of interaction, it will be excluded from the recommendation process.
3. **New user**: When a new user joins the system and has never interacted with any item, the system has no information about their preferences, which prevents the system from generating personalized recommendations.

Scalability Many collaborative filtering methods (such as user-based or item-based approaches) rely on computing pairwise similarity (between pairs of users in the user-based approach, and between pairs of items in the item-based approach). For a system with N users and M items, this results in a computational complexity of $O(N^2)$ or $O(M^2)$, meaning that as the number of users and items increases, the computation becomes more complex, which can increase the computational power needed to generate recommendation in real-time.

Furthermore, collaborative filtering methods often need to update the models with new data (new users and new items), which can be expensive in large-scale systems (Roy and Dutta, 2022).

Sparsity of Interactions If we represent the interactions as a matrix, where on one axis there are the users and on the other axis there are the items, and the rating in the user-item coordinates, then in most of the recommender system cases the matrix will be highly sparse. This means that most of the users only interact with a very small fraction of the universe of items, and vice versa, most of the items are the object of interaction for only a small fraction of users.

This issue can make it difficult for recommendation systems to provide accurate and relevant recommendations, especially for less popular items (Roy and Dutta, 2022).

This issue is closely related to the **popularity bias** problem, which is a common phenomenon in recommender systems. Popularity bias arises when a few very popular items (those with many interactions) dominate the recommendations, while the vast majority of items with fewer interactions are rarely recommended. This means that the system will tend to over-recommend the popular items while barely recommending niche items, further reinforcing their lack of visibility.

In addition to these challenges, the concept of **serendipity** is also important to consider. Serendipity refers to the system's ability to recommend items that are not only relevant to the user but also unexpected and pleasantly surprising. Popularity bias and sparsity can hinder serendipity because the focus on high-interaction items reduces the likelihood of users discovering less obvious but potentially valuable recommendations. Without addressing these issues, the system risks providing recommendations that feel repetitive or overly predictable.

2.2.4 Memory-Based CF

Memory-based collaborative filtering methods compute relevance scores using **user-to-user similarity** or **item-to-item similarity**.

User-to-user similarity

Also known as **user-based collaborative filtering** (Figure 2.3), this approach recommends items based on the preferences of users with similar interaction histories. The assumption is that users with similar past behaviors will have similar future preferences. The objective is to predict $r_{u,i}$, the rating that user u will give to an unrated item i. (Roy and Dutta, 2022).

To find similar users, we compute a similarity score between each pair of users u and v. Common similarity metrics are **Pearson Correlation Coefficient** and **Cosine Similarity**.

Once we have the pairwise similarities, given a user u we find the k-nearest neighbors (i.e., the k users most similar to u based on the similarity scores). We can now compute $r_{u,i}$ as a weighted average of all the ratings $r_{v,i}$ for all the v in u's neighborhood, where the weight is the similarity between u and v.

Each user is represented as a **vector of their interactions** (e.g., rated items), where each dimension corresponds to an item in the universe. Similarity is then computed based on these vectors. For example, if a user has interacted with items 0, 1, and 2 in a system with 10 items, their representation could be a length-10 vector, where only the interacted items have nonzero values—either binary (e.g., presence/absence: $[1, 1, 1, 0, 0, 0, 0, 0, 0, 0]$) or explicit ratings (e.g., $[5, 3, 4, 0, 0, 0, 0, 0, 0, 0]$ on a scale of 1 to 5).

In this standard representation, the order of interactions does not matter—only the presence or strength of interactions is considered. However, certain methods, such as sequential recommenders, explicitly model **temporal dependencies**.

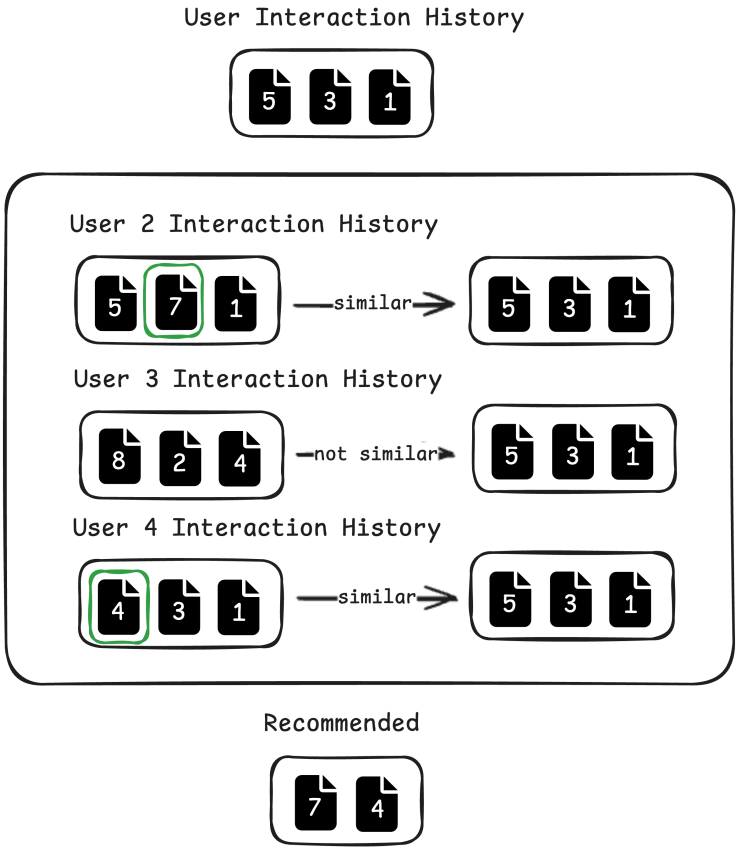


Figure 2.3. How the user-based collaborative filtering recommendation is performed: the user history is compared to the other user's history, and items present in similar histories are recommended.

Item-to-item similarity

Also known as **item-based collaborative filtering** (Figure 2.4), this method makes recommendations based on the similarity between items that the user has already interacted with and other items in the system. The underlying assumption is that if a user has shown interest in an item j, they will likely also be interested in an item i that is highly similar to j.

Each item is typically represented as a **vector of user interactions**, where the vector components correspond to ratings or interaction scores given by different users. For example, an item i may be represented as a vector $i = [r_{u_1,i}, r_{u_2,i}, \dots]$, where $r_{u_k,i}$ denotes the rating (or implicit feedback) of the user u_k for the item i. Similarity between items is then computed based on these vectors using common metrics such as **cosine similarity** or **Pearson correlation coefficient**.

As in user-based collaborative filtering, the goal is to predict $r_{u,i}$, the rating (or relevance)

of an unseen item i for the user u. To do this, we first find the k-**nearest neighbors** of an item i—i.e., the k most similar items that the user has already interacted with. The predicted rating $r_{u,i}$ is then computed as a weighted average of the user's ratings $r_{u,j}$ for these similar items j, where the weights correspond to the similarity scores between i and j.

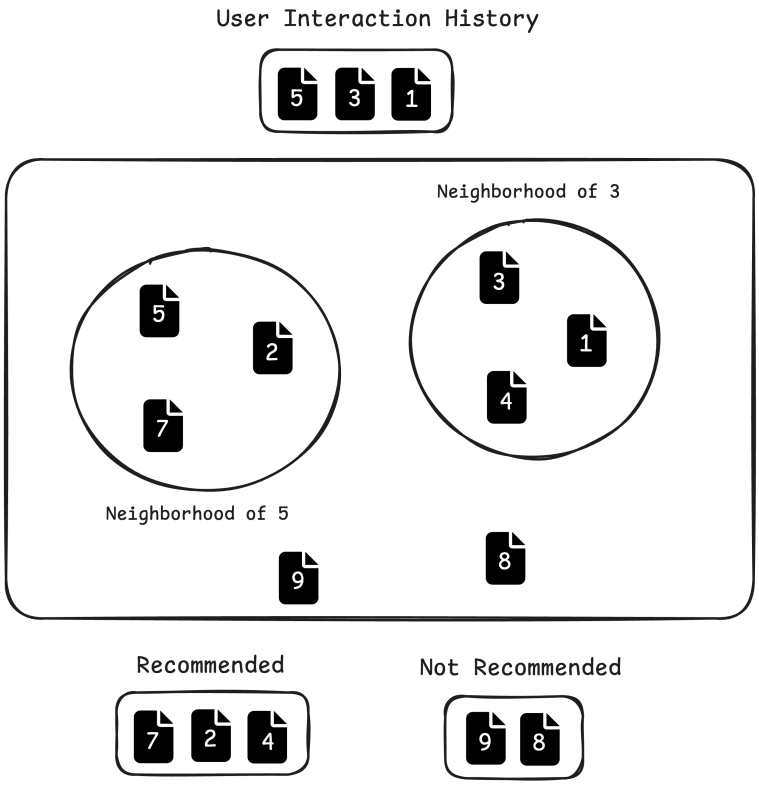


Figure 2.4. How the item-based collaborative filtering recommendation is performed: each item in the user history is compared to each other item, and items in the same neighborhood are recommended.

2.2.5 Model-Based CF

While **memory-based** collaborative filtering directly computes similarities between users or items and makes predictions based on historical interactions, **model-based** collaborative filtering learns a predictive model to estimate a user's rating or preference for unrated items. (Roy and Dutta, 2022)

This approach involves using algorithms such as **Matrix Factorization** (e.g., **Singular Value Decomposition (SVD)**), **K-Nearest Neighbors (KNN)**, and **Neural Networks** to capture underlying patterns in the user-item interactions. By learning latent factors or complex relationships, model-based techniques often provide more scalable and accurate recommendations, especially in large datasets.

2.2.6 Deep Learning-Based Recommendation

The first notable neural network-based method for recommendation was the application of **Restricted Boltzmann Machines (RBMs)** for Collaborative Filtering, proposed by (Salakhutdinov et al., 2007) in *Netflix Prize* [2]. In this work a two-layer RBM is used to model the user-item interaction which is then used to perform recommendation.

In general, deep learning based recommendation techniques involve integrating the distributed item representations learned from auxiliary information such as text, images and acoustic features into collaborative filtering models. (Sun et al., 2019).

Neural Collaborative Filtering (NCF) estimates user preferences using Multi-Layer Perceptrons (MLPs), while in *AutoRec* (Sedhain et al., 2015) and *CDAE* (Wu et al., 2016) the user ratings are predicted using an autoencoder architecture.

2.3 Sequential Recommender Systems

Most user data is **sequential** in nature, such as the history of listened songs on music streaming platforms, purchased items on e-commerce sites, watched movies on streaming platforms, or recently liked videos on social media.

Representing these histories as **one-hot** encoded vectors (or **rating-based** vectors in some cases) discards the temporal aspect of user interactions. Intuitively, recent interactions are more important for the recommendations than older ones, as they better reflect the user's current preferences. Traditional recommender systems, such as the one seen in the previous sections, treat all the items equally, regardless of their position in time, failing to capture this temporal information.

While some adaptations, such as weighting recent interactions more heavily, can make standard **collaborative filtering** methods more time-aware, they still struggle to fully model **temporal dynamics** and evolving user interests. These approaches typically treat interactions as independent events, causing sequential dependencies to not be fully captured.

Therefore, the most widely used approach for sequential recommendation is through the use of deep learning-based sequential models, such as **Recurrent Neural Networks (RNNs)**, **Long Short-Term Memory (LSTMs)** and **Transformer-based** architectures. These models are designed to capture the temporal dependencies in the user history, allowing for a more accurate context-aware recommendation.

For example, if a user has streamed **50 rock songs** followed by **50 hip-hop songs** in their last 100 interactions on a music streaming service, a sequential recommender would likely prioritize recommending hip-hop songs, as they better reflect the user's **current preference**. By using a different **user encoding** and **model architecture**, sequential recommenders can adapt to evolving interests, whereas traditional recommenders treat all past interactions equally and fail to distinguish between **recent preferences** and **older behaviors**.

[2] https://www.netflixprize.com

Note that while SRSs are compatible with explicit item ratings, this setting introduces one potential problem, caused by the fact that the point in time when users provide the rating for a certain item may be different from when the user has interacted with the same item, potentially disrupting the sequential information. (Quadrana et al., 2018)

2.3.1 Formal Definition

By slightly modifying the definition of the recommender system seen at the beginning of this chapter, we can extend it to be coherent with **sequential recommender systems**.

A **sequential recommender system** can be viewed as a function $\mathcal{G}$ that, given an **ordered sequence** of discrete items $\mathcal{S}_u$ that a user u has interacted with **over time**, returns a vector of scores corresponding to each item in the universe $\mathcal{I}$, representing the probability that the user will interact with each item next.

$$\mathcal{G} : \mathcal{S}_u \rightarrow \mathbb{R}^{|\mathcal{I}|}$$

Where:

- $\mathcal{S}_u$ is the ordered sequence of items the user has interacted with up until the present moment. This can include both implicit interactions (e.g., clicks, views) and explicit ratings (e.g., numerical ratings or feedback), allowing the system to account for both types of data in the input sequence.
- $\mathbb{R}^{|\mathcal{I}|}$ is a vector of scores for all items in the universe $\mathcal{I}$, representing the likelihood of future interaction.

Example Suppose the universe is $\mathcal{I} = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$, and a user u has interacted with the ordered sequence $\mathcal{S}_u = [3, 6, 2, 5]$. The sequential recommender system $\mathcal{G}$ computes a score for each item in $\mathcal{I}$:

$$\mathcal{G}(\mathcal{S}_u) = [0.05, 0.1, 0.15, 0.05, 0.05, 0.1, 0.2, 0.15, 0.1, 0.05]$$

Each value represents the predicted probability that the user will interact with the corresponding item next. The system may provide full transparency by returning scores for all items, or limit its output to just the top-k recommendations.

2.3.2 Popular Sequential Recommender Systems

Before the rise of deep learning, sequential recommendation primarily relied on simpler methods such as **Markov Chains (MCs)**. These models predict a user's next action based on their most recent interactions, assuming that future behavior depends only on a short history of previous interactions. Markov Chains are efficient and interpretable; however, they struggle to capture long-term dependencies and complex behavioral patterns, limiting their effectiveness in modeling intricate user preferences (Kang and McAuley, 2018), (Hidasi et al., 2016).

To overcome these limitations, deep learning-based models have been introduced, mirroring the advancements made in traditional recommender systems. These models enhance sequential recommendation by capturing the **time-sensitive** nature of user interactions. Some of the most well-known approaches include **GRU4Rec** (RNN-based), **SASRec** (Self-attention-based), and **BERT4Rec** (BERT-based).

2.4 Counterfactual Explanations for Recommender Systems

In the context of recommender systems, a counterfactual explanation refers to an example where the user's history of interactions is modified in such a way that it would lead to a different recommendation output. Specifically, an **optimal counterfactual example** is the minimal change in the user's interaction history that results in a distinct recommendation compared to the original output.

Since recommender systems typically provide a top-k list of items (e.g., the top-k recommendations), a counterfactual output can be interpreted as one where the new recommendation list differs significantly from the original list. This difference is often measured by a similarity metric, such as **Normalized Discounted Cumulative Gain (NDCG)**, which considers both the rank and relevance of items in the list. The counterfactual recommendation should ideally induce a noticeable change in the ranking of items, reflecting a change in user behavior or preferences.

To generate an effective counterfactual explanation, the perturbation to the user's interaction history must be small but significant enough to produce a measurable impact on the recommendation.

For example, if a user has a history of watching action movies and receives a recommendation for similar movies, a counterfactual explanation could show how their list of recommendations would change if they had rated a few drama movies higher. This helps the user understand how their preferences influence the recommendations they receive, and how subtle changes in behavior can result in different outcomes.

2.4.1 What makes a good counterfactual explanation?

When generating a counterfactual, we would like for it to have a set of properties, which are used as a rule to establish the quality of the goodness of the explanation itself. (Guidotti, 2024)

- **Validity**: the counterfactual $\tilde{x}$ changes the classification output with respect to the original sample x. Formally,
$$\mathcal{F}(x) \neq \mathcal{F}(\tilde{x})$$
- **Minimality (a.k.a. Sparsity)**: No other *valid* counterfactual example $\tilde{x}'$ exists where the number of differing attributes between x and $\tilde{x}'$ is smaller than that between x and $\tilde{x}$. Formally, $\tilde{x}$ is minimal iff
$$\nexists \tilde{x}' \text{ s.t. } |\delta_{x,\tilde{x}'}| < |\delta_{x,\tilde{x}}|$$
where $\delta_{a,b}$ is a function that returns the set of changed features between a and b

- **Similarity (a.k.a. proximity)**: The counterfactual $\tilde{x}$ should be as close as possible to x, with a maximum distance threshold ϵ. Formally, given a distance function d:

$$d(x, \tilde{x}) < \epsilon$$

- **Plausibility**: given a reference population X e.g.,. a dataset), a counterfactual $\tilde{x}$ is *plausible* if its feature values are coherent with those in X, indicating that $\tilde{x}$ could realistically exist within X, because its features would be in the admitted value range of X. Plausibility is a powerful rule since it creates counterfactuals that can be real examples. This increases trust in the explanation, as a realistic example is more compelling than one with unrealistic values.
- **Discriminative power**: the counterfactual example should clearly identify which feature changes affect the decision outcome, in order to figure out the reason why the change in those particular feature changes the output of the classifier. The discriminative power is defined based on a subjective basis that can be difficult to quantify without experiments involving humans.
- **Actionability**: given a set A of actionable features (features that can be mutated), the counterfactual $\tilde{x}$ is defined as *actionable* iff it's generated from x just by mutating actionable features. Formally:

$$\nexists a_i \in \delta_{x,\tilde{x}} \text{ s.t. } \tilde{x}_i = (a_i, v_i) \wedge x_i \notin A$$

- **Causality**: A counterfactual $\tilde{x}$ should reflect realistic causal relationships between features. Changing a feature in $\tilde{x}$ must correspond to a plausible cause-and-effect relationship based on domain knowledge or a learned causal structure. For example, increasing a person's age might reasonably increase their risk of a disease, but modifying an unrelated feature (e.g., their name) should not affect the outcome. Ensuring causal coherence improves the reliability of the counterfactual explanation.
- **Diversity**: let $C = \{\tilde{x}_1, \dots \tilde{x}_k\}$ a set of k *valid* counterfactuals generated from x, then C should be formed by diverse counterfactual, meaning that the difference among all the counterfactuals in C should be maximized.

Chapter 3

Related Works

Explainable AI has received extensive attention in recent years, with numerous methods developed to enhance the interpretability of machine learning models. However, fewer studies specifically focus on counterfactual generation, particularly in sequence-aware models. This gap becomes even more pronounced when introducing the constraint of generating counterfactuals for a specific target label. The following discussion provides an overview of key contributions in the explainable AI field, ranging from general methods to those specifically designed for recommender systems, with an emphasis on counterfactual explanations.

Among the most well-known methods for explainability is **LIME** (Local Interpretable Model-Agnostic Explanations), introduced by Ribeiro et al. (2016). **LIME** generates local surrogate models to approximate a machine learning model's decision boundary by perturbing the input data and learning an interpretable representation. The learning process minimizes a loss function that balances model complexity and fidelity, in order to produce a low-complexity high-fidelity model, increasing the interpretability of the learned model and the accuracy of the generated explanations. The method assigns importance scores to features, allowing users to understand their impact on the prediction. Another widely used approach is **SHAP** (Shapley Additive Explanations), proposed by Lundberg and Lee. **SHAP** unifies multiple additive feature attribution methods and is grounded in cooperative game theory, using Shapley values to ensure consistency and local accuracy when distributing feature importance scores.

Beyond feature attribution methods, rule-based approaches have also been explored. **LORE** (Local Rule-Based Explanations) (Guidotti et al., 2018) constructs decision trees based on a locally generated dataset consisting of "good" and "bad" instances—points similar to the source instance but differing in their classification label. Using a genetic algorithm, **LORE** selects instances that optimize a fitness function which includes the distance from the source instance and the distance from the source label, ultimately providing human-interpretable if-then rules extracted from the learned decision tree for model explanations.

Shifting focus to counterfactual explanations, several works have addressed the need for actionable and feasible counterfactuals. **FACE** (Feasible and Actionable Counterfactual

Explanations) (Poyiadzi et al., 2020) emphasizes feasibility and actionability. Feasibility ensures that counterfactual instances align with real-world data distributions, while actionability guarantees that suggested changes are practical (e.g., modifying financial behavior rather than age). **FACE** achieves this by constructing a kNN-graph where nodes represent similar instances and edges encode spatial density, identifying counterfactuals through shortest-path algorithms.

MACE (Model-Agnostic Counterfactual Explanations) (Karimi et al.) formulates counterfactual generation as a Boolean satisfiability problem (SAT). By converting both the model and distance metric into logical expressions, **MACE** employs an SMT solver to find counterfactuals that satisfy both conditions. This method is particularly powerful due to its model-agnostic nature, though it requires access to the model's full code in order to convert it to a logical formula, making it a gray-box approach.

As seen for **LORE** (Guidotti et al., 2018), genetic algorithms have also been leveraged for counterfactual generation. **CERTIFAI** (Counterfactual Explanations for Robustness, Transparency, Interpretability, and Fairness of AI models) (Sharma et al., 2020), employs a model-agnostic custom genetic algorithm to generate a set of potential counterfactual examples near the source point's neighborhood by optimizing a fitness function that balances similarity to the original instance and the predicted label of the counterfactual, while **GeCo** (Genetic Counterfactuals) (Schleich et al., 2021) optimizes the genetic algorithm iteration when the model access is white-box, making the counterfactual generation in real-time.

DiCE (Diverse Counterfactual Explanations) (Mothilal et al., 2020), generates multiple diverse counterfactuals using gradient-based optimization, making it a gray-box approach dependent on the model's gradient information.

While the aforementioned methods focus on general counterfactual explanations, some works have been specifically tailored for recommender systems. **ACCENT** (Action-based Counterfactual Explanations for Neural Recommenders for Tangibility) (Tran et al., 2021) introduces a novel approach to counterfactual reasoning in recommendation systems, specifically for **neural recommenders**. Unlike traditional methods that rely on attention mechanisms or metadata-based explanations, **ACCENT** generates counterfactuals based on a user's own past interactions. It extends influence functions to estimate the importance of user actions and iteratively identifies the minimal set of actions that, if removed, would alter the recommendation outcome. **CauseRec** (Counterfactual User Sequence Synthesis for Sequential Recommendation) (Zhang et al., 2021) utilizes counterfactual examples as a data augmentation technique with contrastive learning to address the inherent sparsity of recommender system datasets. By leveraging counterfactual reasoning, CauseRec enhances user representations and improves recommendation accuracy.

Chapter 4

Method

In this work, we address the problem of **explanations through targeted and untargeted, categorized and uncategorized counterfactual examples in sequential recommender systems**.

We aim to build an **explainer** that takes a **black-box model** (also referred to as an **oracle**), a **source sequence**, and optionally a **target category**—if the setting is **targeted**—and generates a modified sequence. In the **targeted** setting, the output sequence is similar to the source sequence but classified with the target label by the black-box model. In the **untargeted** setting, the explainer instead produces a sequence that leads to a different classification than the original.

The desired explainer is classified as:

- **Black-box**, because the internals of the model do not need to be known. The explainer operates based on the model's output—which is ideally the raw logits—though it will also work with the final top-k ranking.
- **Data specific**, as it is designed specifically for **sequences of discrete items**.
- **Exogenous**, since the explanation consists of a **sequence** made up of items taken from a broader universe of items. While the explanation is not guaranteed to belong to the original dataset, it does not need to be restricted to it.

The generated counterfactuals are expected to satisfy the following properties:

- **Validity**, meaning that counterfactual's label will be the target label if the setting is targeted, or a different label if the setting is untargeted.
- **Actionability**, indicating that the user can feasibly modify their sequence of interactions to achieve the counterfactual label. This means that the changes made to the original sequence are **realistic** and can be **implemented** within the user's context or constraints.

- **Similarity**, ensuring that the counterfactual sequence is similar to the original source sequence.
- **Plausibility**, meaning that the counterfactual is not only similar to the original sequence but also realistic and plausible, i.e., it makes sense within the domain and context of the recommender system.

However, the following properties are not necessarily satisfied:

- **Minimality**, the counterfactual sequence may not be the smallest possible modification required to change the label, i.e., the solution is not necessary **optimal**.
- **Discriminative power**, the counterfactual may not always highlight the most discriminative factors that differentiate the original and counterfactual outcomes.
- **Diversity**, even if not explicitly evaluated nor tested in the current work, both the GENE and PACE methods can generate multiple counterfactuals from a single source sequence, but their diversity is not guaranteed.
- **Causality**: Ensuring that changes in the input sequence have a truly causal impact on the model's decision—rather than exploiting spurious correlations—is inherently challenging. Since this property has not been explicitly tested, we cannot determine whether it is satisfied. Moreover, verifying causality would require a more in-depth analysis that goes beyond the scope of this work.

4.1 The Four Settings

We recall that a **counterfactual explainer** is an algorithm $\mathcal{F}$ that, given an input instance x, generates a counterfactual example $\tilde{x}$ such that the model's decision on $\tilde{x}$ differs from its decision on x (i.e., $\mathcal{F}(\tilde{x}) \neq \mathcal{F}(x)$), while ensuring that the difference between x and $\tilde{x}$ is minimal.

Since we are working within the framework of sequential recommender systems, we define the problem using the expected structure of sequential recommendation models.

Let $\mathcal{U} = \{u_1, u_2, \ldots, u_{|\mathcal{U}|}\}$ be the set of users and $\mathcal{I} = \{i_1, i_2, \ldots, i_{|\mathcal{I}|}\}$ be the universe of items. Each user u is associated with a **history of interactions**, represented as an **ordered sequence** of items:

$$\mathcal{I}_u = (i_1^u, i_2^u, \ldots, i_j^u, \ldots, i_{|\mathcal{I}_u|}^u)$$

where each i_j^u represents an interaction event (e.g., user u viewing, purchasing, or engaging with item i).

4.1.1 Untargeted-Uncategorized

In the untargeted and uncategorized setting, a **counterfactual sequence** is a hypothetical sequence $\tilde{\mathcal{I}}_u$ derived from the original sequence $\mathcal{I}_u$ by modifying some **items** to alter the recommendation outcome. The optimal counterfactual sequence is the closest sequence to $\mathcal{I}_u$ that results in a different prediction from the recommender system $\mathcal{G}$.

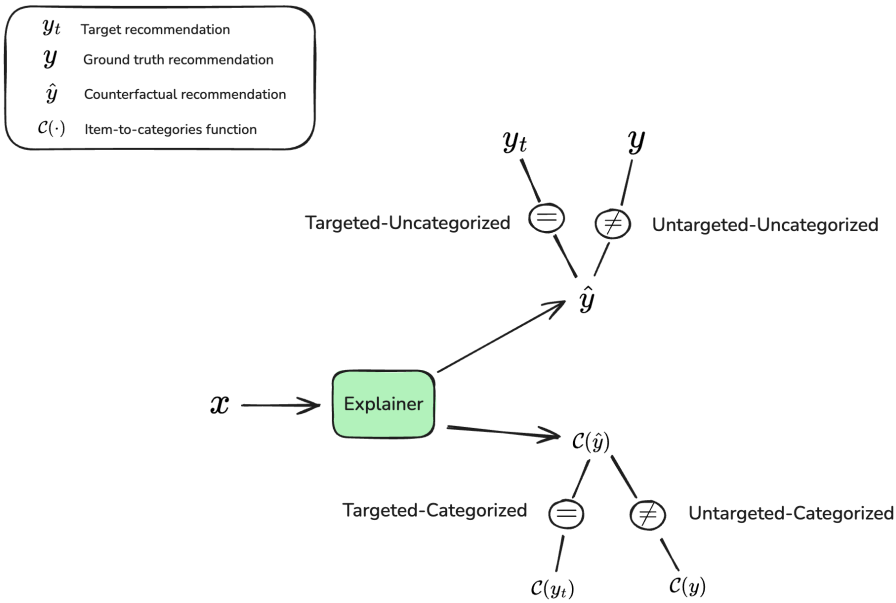


Figure 4.1. A diagram showing the different objective for a counterfactual to be valid in each setting.

Formally, given an original interaction sequence $\mathcal{I}_u = (i_1, i_2, \ldots, i_n)$, a counterfactual sequence $\tilde{\mathcal{I}}_u = (\tilde{i}_1, \tilde{i}_2, \ldots, \tilde{i}_n)$ satisfies:

$$\mathcal{G}(\tilde{\mathcal{I}}_u) \neq \mathcal{G}(\mathcal{I}_u)$$

while minimizing the distance between $\mathcal{I}_u$ and $\tilde{\mathcal{I}}_u$. That is, the optimal counterfactual sequence solves the following optimization problem:

$$\tilde{\mathcal{I}}_u = \arg\min_{\tilde{\mathcal{I}}_u} \delta(\mathcal{I}_u, \tilde{\mathcal{I}}_u) \quad \text{subject to} \quad \mathcal{G}(\tilde{\mathcal{I}}_u) \neq \mathcal{G}(\mathcal{I}_u)$$

where:

- $\delta(\mathcal{I}_u, \tilde{\mathcal{I}}_u)$ is a distance metric, such as **edit distance**, measuring the similarity between sequences.
- $\mathcal{G}(\cdot)$ is the sequential recommender system, which predicts the next recommended item based on the given interaction history.

For simplicity, in this definition we are assuming a *top-1* recommendation setting, where the recommender outputs only the most likely next item. However, in practical applications, recommender systems often return a ranked list of *top-k* items. Our approach is fully compatible with this *top-k* setting via the proposed **counterfactual@k** framework (Section 4.2).

Finding the **optimal** counterfactual sequence is an NP-hard problem, as shown in Tsirtsis and Gomez Rodriguez (2020), proven by using a reduction to the Set Cover problem.

4.1.2 Untargeted-Categorized

Experiments and prior research (e.g., Betello et al. (2023)) reveal that recommender models are highly sensitive to recent interactions.

This causes counterfactuals to quickly be found by simply modifying—or removing—the last user interaction. While valid in theory, this counterfactual doesn't always provide a valuable explanation to the final user.

To address this, we incorporate **item categories** into counterfactual explanations. Instead of merely changing recommendations, we seek counterfactuals that alter the category of the recommended item. Given a function $\mathcal{C}$ that maps items to categories, we define a counterfactual sequence such that:

$$\mathcal{C}(\mathcal{G}(\tilde{\mathcal{I}}_u)) \neq \mathcal{C}(\mathcal{G}(\mathcal{I}_u))$$

Since categories are predefined within the dataset, this function acts as a simple lookup.

Using categories makes the recommender less sensitive to the last positions due to the higher sparsity of solutions, and it also provides the final user with a more valuable explanation.

For example, if the source sequence's recommendation is a horror movie and the explainer proposes a counterfactual where the recommendation is another movie but still in the horror genre, the explanation has less value than proposing a counterfactual where the recommendation is of a different movie genre.

Notably, an item may belong to multiple categories, making direct comparisons between outputs more complex. This challenge, particularly in the context of *top-k* recommendations, is discussed and solved with the proposed **counterfactual@k** framework.

4.1.3 Targeted-Uncategorized

To enhance explanation, we can introduce a **target** y_t, which further constrains valid solutions to those yielding the target label. Differently from the untargeted setting, we aim to minimally modify $\mathcal{I}_u$ to achieve a specific target recommendation, rather than just any alternative prediction. Formally we want:

$$\hat{\mathcal{I}}_u \quad \text{such that} \quad \mathcal{G}(\tilde{\mathcal{I}}_u) = y_t$$

The properties that define a good counterfactual remain the same as in the untargeted case.

Finding a valid and optimal counterfactual is more challenging in the targeted setting due to the higher sparsity of solutions.

4.1.4 Targeted-Categorized

The only difference the **targeted-categorized** setting has from its uncategorized counterpart is the fact that the target is a category, expressed as $\mathcal{C}(y_t)$, and we want:

$$\hat{\mathcal{I}}_u \quad \text{such that} \quad \mathcal{C}(\mathcal{G}(\tilde{\mathcal{I}}_u)) = \mathcal{C}(y_t)$$

	Targeted	**Untargeted**
Uncategorized	$\mathcal{G}(\tilde{\mathcal{I}}_u) = y_t$	$\mathcal{G}(\tilde{\mathcal{I}}_u) \neq \mathcal{G}(\mathcal{I}_u)$
Categorized	$\mathcal{G}(\tilde{\mathcal{I}}_u) = y_t, \quad \mathcal{C}(\mathcal{G}(\tilde{\mathcal{I}}_u)) \neq \mathcal{C}(\mathcal{G}(\mathcal{I}_u))$	$\mathcal{C}(\mathcal{G}(\tilde{\mathcal{I}}_u)) \neq \mathcal{C}(\mathcal{G}(\mathcal{I}_u))$

Table 4.1. A recap of the validity conditions for a counterfactual in the different settings

4.2 Counterfactual@k Framework

When defining the conditions for a datapoint to be *counterfactual*, we interpreted the inequality between labels (expressed as $y \neq \hat{y}$) as "the two labels are different." This interpretation holds if the recommender system returns the **most probable item** a user will interact with—i.e., the prediction is *top-1*. In this case, both the original label y and the counterfactual label $\hat{y}$ are integers, making direct comparison straightforward.

However, recommendations are not always reliable when considering only the next item. A more robust approach is *top-k*, where the ranking of the most probable next k items is considered. In this setting, both the original and counterfactual labels are not single integers but rather **ordered lists**—i.e., rankings—of length k.

Interpreting the inequality $y \neq \hat{y}$ as "the two ordered lists are different" is no longer valid. Two highly similar—but not equal—rankings would be treated as entirely different, just as two completely distinct rankings would be. Instead of a strict inequality, we measure the similarity between the ordered lists and apply a threshold t to the similarity score $\delta(y, \hat{y})$ to obtain a boolean value. Formally, in the counterfactual@k, we redefine the equality as following:

$$y = \hat{y} \iff \delta(y, \hat{y}) > t$$

For assessing ranking quality, the most suitable metric is the **Normalized Discounted Cumulative Gain** (NDCG).

The NDCG consists of the Cumulative Gain (CG), which sums all item relevance scores—values indicating how relevant a product is for the user, obtained in various ways. To account for item order, we introduce the **Discounted** component (DCG), where each term in the summation is weighted by a factor that decreases as the item's position in the ranking increases. To ensure DCG falls between 0 and 1, we **Normalize** it (NDCG) by computing the ratio of DCG to the **Ideal Discounted Cumulative Gain** (IDCG), which represents the DCG of a perfect ranking.

Formally:

$$\text{DCG} = \sum_{i=1}^{k} \frac{rel_i}{\log_2(i+1)}$$

and

$$\text{NDCG} = \frac{\text{DCG}}{\text{IDCG}}$$

The relevance scores are computed depending on the particular setting.

4.2.1 Untargeted-Uncategorized

We first address the simplest case, where the counterfactual is generated without a target and the model output is a ranking of items (untargeted, uncategorized).

To use NDCG as a similarity measure between the source sequence's rank r_{gt} and the counterfactual rank r_{pred}, we set the IDCG as the DCG of the source sequence's rank with itself—i.e. $\text{IDCG} = \text{DCG}(r_{\text{gt}}, r_{\text{gt}})$. A higher DCG between the source sequence's rank and the counterfactual's rank indicates greater similarity to the perfect ranking.

When deciding how to compute relevance scores, we consider the following:

1. If the two ranks are identical, the score should be 1. This is ensured by the IDCG computation logic.
2. If two ranks contain the same items but in a different order, where there are no pairwise matches, the relevance score for each item should be proportional to the **positional difference** between the item in the first and second ranks.
3. If the two ranks share no common items, the score should be 0.

To achieve these properties, we compute rel_i as the reciprocal of the absolute positional difference between item i in r_{gt} and its position j in r_{pred}:

$$rel_i = \frac{1}{|i - j| + 1}$$

where

$$j := \min\{j \mid r_{\text{gt}}[i] = r_{\text{pred}}[j]\}$$

We call the verification of a potential counterfactual's ranking against the source sequence's ranking using a top-k approach **Counterfactual@k**. This is motivated by the fact that different k can yield different scores, above or below the threshold, meaning that a counterfactual can be valid at one value of k, but not valid at another value of k.

The choice of k is crucial and depends on the application context. Lower values of k are generally preferred in scenarios where precision is critical, such as search engines, medical diagnosis systems, or high-stakes recommendation systems where only the top results matter. In these cases, a counterfactual must maintain high relevance within the top few positions to be considered valid.

On the other hand, higher values of k are more suitable for applications where diversity and exploration are important. For example, in content recommendation, music streaming, or e-commerce browsing, users often consider a broader range of options rather than just the top-ranked items. In such cases, a counterfactual can still be relevant even if it ranks lower in the list, making a higher k more appropriate for evaluation.

Example Let's make an example of execution to showcase the practical computation.

- Let $\mathcal{U} = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ be the universe of items, represented by their IDs.
- Let $x = [1, 4, 6, 7]$ be the source sequence and $\hat{x} = [1, 5, 6, 8]$ the counterfactual sequence.
- Let $y = [2, 5, 3]$ be the ground truth ranking and $\hat{y} = [2, 10, 3]$ the counterfactual ranking both cutout at $k = 3$.

We compute the relevance scores using the aforementioned formula:

$$rel_i = \frac{1}{|i - j|}$$

Applying this to each position in the ranking:

- $r_{\text{gt}}[1] = 2, j \quad s.t. \quad r_{\text{pred}}[j] = 2$ is $j = 1$, so $\text{rel}_1 = 1$
- $r_{\text{gt}}[2] = 5, j \quad s.t. \quad r_{\text{pred}}[j] = 5$ does not exist , so $\text{rel}_2 = 0$
- $r_{\text{gt}}[3] = 3, j \quad s.t. \quad r_{\text{pred}}[j] = 3$ is $j = 3$, so $\text{rel}_3 = 1$ Thus, the cumulative gain vector is:

$$CG = [1, 0, 1]$$

Computing the discounted cumulative gain:

$$DCG = \left[\frac{1}{\log_2(2)}, \frac{0}{\log_2(3)}, \frac{1}{\log_2(4)}\right] = [1, 0, 0.5]$$

Since the ideal ranking perfectly matches itself, the ideal cumulative gain remains:

$$IDCG = [1, 1, 1]$$

Applying the same discounting:

$$IDCG = [1, 0.631, 0.5]$$

Final computation of $NDCG$:

$$NDCG = \frac{1.5}{2.131} = 0.7039$$

4.2.2 Targeted-Uncategorized

When dealing with the **targeted** setting, there isn't the need to compute the positional difference between the ground truth item and the predicted item in the two rankings. Differently from the targeted-categorized setting (Section 4.2.4) we surely won't have duplicates in the rankings, hence the score has to be proportional to the absolute position of the desired target in the predicted ranking. We can do this by just setting the ideal

score to 1, motivated by the fact that the perfect ranking is the one that puts the target item in the first place, obtaining a DCG of 1.

$$rel_i = \mathbb{1}_{r_{\text{gt}}[i]=r_{\text{pred}}[i]}$$

The higher the position of the target in the ranking, the higher the NDCG will be.

Example

- Let $\mathcal{U} = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ be the universe of items, represented by their IDs.
- Let $x = [1, 4, 6, 7]$ be the source sequence and $\hat{x} = [1, 5, 6, 8]$ the counterfactual sequence.
- Let $\hat{y} = [2, 10, 3]$ be the counterfactual ranking cutout at $k = 3$.
- Let $y_t = 10$ be the target label.

We create the ground truth ranking by filling a vector of the same length of $\hat{y}$ with the target item $y_t = 10$.

$$y_{gt} = [10, 10, 10]$$

The relevance scores are computed based on exact matches:

$$rel_i = \mathbb{1}_{r_{\text{gt}}[i]=r_{\text{pred}}[i]}$$

Since $r_{\text{pred}} = [2, 10, 3]$:

- $r_{\text{pred}}[1] \neq 10 \Rightarrow rel_1 = 0$
- $r_{\text{pred}}[2] = 10 \Rightarrow rel_2 = 1$
- $r_{\text{pred}}[3] \neq 10 \Rightarrow rel_3 = 0$

  Thus:

$$CG = [0, 1, 0]$$

Computing discounted cumulative gain:

$$DCG = [0, \frac{1}{\log_2(3)}, 0] = [0, 0.631, 0]$$

Final computation of $NDCG$:

$$NDCG = \frac{0.631}{1} = 0.631$$

4.2.3 Untargeted-Categorized

In the categorized setting, the situation is more complex since rankings can no longer be represented as simple lists of integers. Each item can belong to multiple categories, meaning that passing an item through the item-to-category mapping $\mathcal{C}(\cdot)$ will yield a set of category identifiers (represented as integers). Mapping each item in the top-k ranking results in a **list of sets of integers**, where each set has a minimum length of 1.

Using the **equal** operator for comparison is suboptimal because two items with partial category overlap would be considered **as different as two items with no category overlap at all**. To address this, relevance should be weighted according to the degree of category overlap between the two compared sets.

Depending on the setting (untargeted vs. targeted), the computation of the **intersection-weighted relevance score** follows slightly different approaches.

In the **untargeted** case we use the normalized intersection as weight for the relevance score.

$$rel_i = \underbrace{\frac{r_{\text{gt}}[i] \cap r_{\text{pred}}[j]}{\min(|r_{\text{gt}}|, |r_{\text{pred}}|)}}_{\text{normalized intersection}} \cdot \underbrace{\frac{1}{|i-j|}}_{\text{positional relevance}}$$

where

$$j := \min\{j \mid r_{\text{gt}}[i] \cap r_{\text{pred}}[j] \neq \emptyset\}$$

This yields a value that is directly proportional to the ratio of the intersection size with respect to the maximum intersections size; while still being inversely proportional to the distance between the current set in gt and the first matching set in preds, where two sets match if their intersection is non-empty.

Example Let $\mathcal{U} = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ be the universe of items, represented by their IDs.

- Let $\{c_1, c_2, c_3, c_4, c_5\}$ be the set of possible categories.
- Let $\mathcal{C}$ be the function that maps item IDs into set of categories.
- Let $x = [1, 4, 6, 7]$ be the source sequence and $\hat{x} = [1, 5, 6, 8]$ the counterfactual sequence.
- Let $y = [2, 5, 3]$ be the ground truth ranking and $\hat{y} = [2, 10, 3]$ the counterfactual ranking both cutout at $k = 3$.
- Let $\mathcal{C}(y) = [\{c_1, c_2\}, \{c_3\}, \{c_1\}]$ the ground truth ranking and $\mathcal{C}(\hat{y}) = [\{c_1, c_2\}, \{c_5\}, \{c_1\}]$ the counterfactual ranking, both in terms of categories.

We compute the cumulative gain with the categorized formula. For each index $i \in (1, 3)$, we determine the best matching index j in the predicted ranking based on category overlap. Using the normalized intersection-weighted relevance formula:

$$rel_i = \frac{|r_{\text{gt}}[i] \cap r_{\text{pred}}[j]|}{\min(|r_{\text{gt}}[i]|, |r_{\text{pred}}[j]|)} \cdot \frac{1}{|i-j|}$$

Computing j for each i:

- For $i = 1$: $r_{\text{gt}}[1] = \{c_1, c_2\}$, best matching j is 1, since $r_{\text{pred}}[1] = \{c_1, c_2\}$ (perfect match), yielding $rel_1 = 1$.
- For $i = 2$: $r_{\text{gt}}[2] = \{c_3\}$, best matching j is 2, since $r_{\text{pred}}[2] = \{c_5\}$ (no match), yielding $rel_2 = 0$.
- For $i = 3$: $r_{\text{gt}}[3] = \{c_1\}$, best matching j is 3, since $r_{\text{pred}}[3] = \{c_1\}$ (perfect match), yielding $rel_3 = 1$. Thus, the cumulative gain vector is:

$$CG = [1, 0, 1]$$

Then we compute the discounted cumulative gain:

$$DCG = \left[\frac{1}{\log_2(2)}, \frac{0}{\log_2(3)}, \frac{1}{\log_2(4)}\right] = [1, 0, 0.5]$$

Since the ideal ranking would match perfectly with itself, the ideal cumulative gain remains:

$$IDCG = [1, 1, 1]$$

Applying the same discounting:

$$IDCG = [1, 0.631, 0.5]$$

Computing the final $NDCG$ score:

$$NDCG = \frac{1.5}{2.131} = 0.7039$$

4.2.4 Targeted-Categorized

The relevance score in the categorized targeted case can be determined based on how strict we want the target condition to be:

1. On the less strict end of the spectrum, relevance is perfect when just **one item overlaps** between the two sets:

$$rel_i = \mathbb{1}_{r_{\text{gt}}[i] \cap r_{\text{pred}}[i] \geq 1}$$

2. On the stricter end, relevance is perfect only when **all items overlap**, which is equivalent to using the **equal operator**:

$$rel_i = \mathbb{1}_{r_{\text{gt}}[i] = r_{\text{pred}}[i]}$$

3. A middle-ground approach normalizes the intersection size between sets:

$$rel_i = \frac{r_{\text{gt}}[i] \cap r_{\text{pred}}[j]}{\min(|r_{\text{gt}}|, |r_{\text{pred}}|)}$$

In this thesis, since we only require the final recommendation to belong to the target category, we adopt the **first approach**. This means that we do not account for additional category memberships—only whether the item belongs to the target category.

Example

- Let $\mathcal{U} = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ be the universe of items, represented by their IDs.
- Let $\{c_1, c_2, c_3, c_4, c_5\}$ be the set of possible categories.
- Let $\mathcal{C}$ be the function that maps item IDs into set of categories.
- Let $x = [1, 4, 6, 7]$ be the source sequence and $\hat{x} = [1, 5, 6, 8]$ the counterfactual sequence.
- Let $y = [2, 5, 3]$ be the ground truth ranking and $\hat{y} = [2, 10, 3]$ the counterfactual ranking both cutout at $k = 3$.
- Let $\mathcal{C}(\hat{y}) = [\{c_1, c_2\}, \{c_5\}, \{c_1\}]$ be the counterfactual ranking in terms of categories.
- Let $y_t = c_5$ the target category.

Same as the targeted-uncategorized setting, we repeat the target k times, obtaining the perfect ranking:

$$y_{gt} = [c_5, c_5, c_5]$$

Using the least strict relevance criterion ($rel_i = \mathbb{1}_{r_{\text{gt}}[i] \cap r_{\text{pred}}[i] \geq 1}$):

- $r_{\text{pred}}[1] = \{c_1, c_2\}$ has no overlap with $c_5 \Rightarrow rel_1 = 0$
- $r_{\text{pred}}[2] = \{c_5\}$ has perfect overlap $\Rightarrow rel_2 = 1$
- $r_{\text{pred}}[3] = \{c_1\}$ has no overlap with $c_5 \Rightarrow rel_3 = 0$ Thus:

$$CG = [0, 1, 0]$$

Computing discounted cumulative gain:

$$DCG = [0, \frac{1}{\log_2(3)}, 0] = [0, 0.631, 0]$$

The ideal ranking has:

$$IDCG = [1, 0.631, 0.5]$$

Final computation of $NDCG$:

$$NDCG = \frac{0.631}{2.131} = 0.296$$

A recap of the NDCG relevance score computation for all the settings is detailed in Table 4.2

	Targeted	Untargeted
Uncategorized	$rel_i = \mathbb{1}_{r_{\text{gt}}[i]=r_{\text{pred}}[i]}$	$rel_i = \frac{1}{\lvert i-j \rvert}$ where $j := \min\{j \lvert r_{\text{gt}}[i] = r_{\text{pred}}[j]\}$
Categorized	$rel_i = \mathbb{1}_{r_{\text{gt}}[i] \cap r_{\text{pred}}[i] \geq 1}$	$rel_i = \frac{r_{\text{gt}}[i] \cap r_{\text{pred}}[j]}{\min(\lvert r_{\text{gt}} \rvert, \lvert r_{\text{pred}} \rvert)} \cdot \frac{1}{\lvert i-j \rvert}$ where $j := \min\{j \lvert r_{\text{gt}}[i] \cap r_{\text{pred}}[j] \neq \emptyset\}$

Table 4.2. A recap of the relevance score computation for all the settings.

4.3 Proposed Counterfactual Generation Methods: GENE and PACE

This work introduces two correlated counterfactual explanation approaches designed for sequential recommender systems. The first is **GENE** (**GEN**etic **C**ounterfactual **E**xplanations), which uses a genetic algorithm to generate counterfactual examples. The second is **PACE** (**P**ath-search **A**utomata **C**ounterfactual **E**xamples), which builds on GENE to create an automaton that functions as a local surrogate model for both counterfactual interpretation and generation.

Let's call the source point x and its outcome y. At high level, the outline of the **GENE** algorithm is the following: we generate a dataset consisting of **good** and **bad** points. The **good** points represent sequences where the **condition** is satisfied, while the **bad** points represent sequences where the condition is not satisfied. The condition for a point to be counterfactual varies depending on the setting. When using GENE, we generate only the **good dataset**, which contains the counterfactual examples that **satisfy the condition**. We select the k best counterfactuals that are most similar to the source sequence x according to a chosen distance metric (e.g., edit distance).

PACE extends GENE by considering both the **good** and **bad** parts of the dataset to learn an automaton that distinguishes between them. Since the good and bad datasets include points close to the source sequence, the automaton functions as a **local surrogate model** of the **black-box sequential model** used to generate the labels.

Additionally, the automaton is augmented with the ability to include possible edits (additions and deletions), inspired by the work of De Giacomo et al. (2017). This enhanced automaton represents the counterfactual state of the sequence at each position, with a certain level of accuracy..

Finally, by using a variation of the A* algorithm, referred to as **Constrained** $A^\star$, we

can find the best path of edits that leads to the counterfactual nearest to the source sequence.

4.4 GENE: GENetic Counterfactual Explanations

GENE is a counterfactual explanation approach designed for **sequential recommender systems** (diagram in Figure 4.2). It adapts the **LORE** framework (Guidotti et al., 2018) to work with ordered sequences of discrete items. The core idea behind GENE is to generate counterfactual examples by using a **genetic algorithm**. This allows us to explore the space of possible sequences efficiently, searching for those that best satisfy the counterfactual conditions.

GENE operates by iteratively modifying a given **source sequence** through a set of mutation operations, optimizing toward a desired counterfactual outcome. Unlike LORE, which works with tabular data, GENE is specifically tailored for **ordered sequences of discrete items without repetition**, making it a more suitable approach for sequential recommendation tasks.

4.4.1 Genetic Algorithm

A genetic algorithm is a **metaheuristic**—a high-level procedure inspired by the process of **natural selection** and based on **evolution**. It is used for optimization tasks, aiming to minimize (or maximize) a function known as the **fitness function**.

The high-level steps of a genetic algorithm are as follows:

1. **Initialize Population**: Start with a population of size N (usually represented as an array). The initial population can be chosen randomly or through other methods.
2. **Evaluate Fitness**: Evaluate the *fitness function* for each individual in the population, producing an array of fitness scores.
3. **Mutation**: *Mutate* a percentage of the current population. Mutation involves editing an individual, typically by altering one or more values in the array.
4. **Crossover**: Perform *crossover* for a subset of individual pairs. Crossover is an operation that combines the traits of both individuals to create new individuals.
5. **Evaluate New Population**: Re-evaluate the fitness function for each individual in the new population.
6. **Selection**: Select the top-K individuals from the new population based on their fitness values (the highest or lowest, depending on whether we are maximizing or minimizing the fitness function). These selected individuals become the *new population*.
7. **Repeat**: Repeat steps 2 through 6 for a certain number of generations.

With each generation, only individuals with the highest (or lowest) fitness values will "survive" and reproduce, while others are discarded. This process leads to a final population

that optimizes the fitness function.

Tailoring the genetic algorithm for sequences of items

LORE—the method proposed by Guidotti et al. (2018)—, generate **rule-based explanations** by learning a Decision Tree, which acts as a local surrogate model. The Decision Tree is trained using a dataset of **good** and **bad** points, starting from a **source point** x.

The idea behind dataset generation is to create points that approximate the **neighborhood** of x, selecting only those points that are very similar to x based on a certain distance metric. Since the Decision Tree is trained on the neighborhood of x, it can approximate the decision-making process of the black-box model in that local region.

If y is the label of the source point (the output of the black-box model), the **good** points will contain those that are similar to x and share the same label y. On the other hand, the **bad** points are also similar to x, but have different labels $y_1, y_2, ...$ (these are counterfactual examples, as defined earlier).

While LORE was originally developed for tabular data, the main modification needed to apply it to sequential data is in the **mutation** operation. Since **selection** and **crossover** naturally work with sequences of items, they can remain unchanged from LORE.

Given a source vector x and a universe of items $\mathcal{I}$, we define **six mutations** tailored for ordered sequences of discrete items without repetition:

1. **Replace**: Randomly choose an index i within the bounds of x and a random item $z \in \mathcal{I}$ that is not already in x. Replace the item at index i in x with z, i.e., $x[i] = z$.
2. **Add**: Randomly choose an index i within the bounds of x and a random item $z \in \mathcal{I}$ that is not already in x. Insert z into x at position i. The insertion involves adding a slot at the end of x, shifting each element after the i-th position to the right ($x[j+1] = x[j] \quad \forall j \geq i$), and placing the new item in the i-th position ($x[i] = z$).
3. **Delete**: Randomly choose an index i within the bounds of x and delete the value at position i, shifting all elements after it one position to the left.
4. **Swap**: Randomly choose two indices (i, j) and swap the items at these indices, i.e., $x[i] = x[j]$.
5. **Shuffle**: Randomly choose two indices (i, j) such that $i < j$, and shuffle the subsequence of x from index i to j, i.e., $x[i:j] = \text{shuffle}(x[i:j])$.
6. **Reverse**: Randomly choose two indices (i, j) such that $i < j$, and reverse the subsequence of x from index i to j, i.e., $x[i:j] = \text{reverse}(x[i:j])$.

The *Swap*, *Shuffle*, and *Reverse* operations can be highly disruptive if the indices chosen are far apart, potentially resulting in a significant alteration of the sequence. To prevent this, a *maximum offset* parameter is introduced. This parameter controls the maximum allowed distance between the two indices, expressed as a fraction of the sequence length. For

instance, if the *maximum offset* is set to 0.2 and the sequence length is 50, the maximum distance between i and j would be 10.

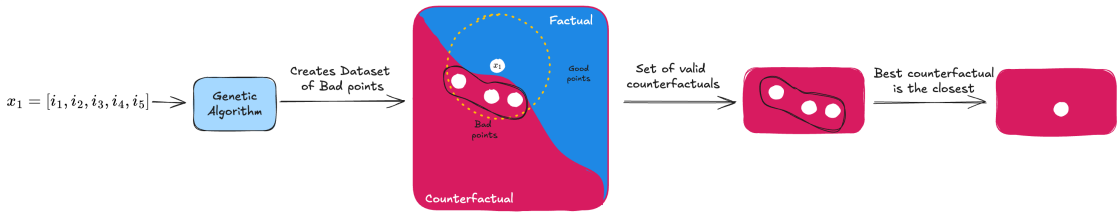


Figure 4.2. Diagram showing GENE's pipeline, generating a counterfactual starting from the original sequence.

4.5 PACE: Path-search Automata Counterfactual Examples

The **Path-search Automata Counterfactual Examples (PACE)** algorithm generates counterfactual explanations by modeling the search for minimal edits as a path-search problem over an augmented deterministic finite automaton (DFA). Unlike traditional approaches relying on heuristic search or gradient-based methods, PACE constructs a deterministic constraint automaton $\mathcal{A}^+$ that encodes both the structure of the input space and the decision boundaries of a black-box model $\mathcal{F}$.

By incorporating repair propositions—special transitions representing edits—the automaton efficiently explores possible counterfactual sequences without explicitly enumerating all edits. The algorithm finds the shortest sequence of edits that transforms an input into a counterfactual, ensuring minimal modifications.

The counterfactual search is performed using a **constrained** variant of the A* algorithm, where constraints dynamically restrict the considered edges based on the symbols read by the automaton. This ensures that the algorithm navigates the state space coherently, identifying the shortest valid edit sequence that transforms an input into a counterfactual.

The automaton is learned using the dataset of good and bad points generated by GENE as training data. The learned DFA is then augmented by rules defined in the work by De Giacomo et al. (2017), which allows the automata to encode possible sequence edits. In De Giacomo et al. (2017) they use the automata as a solution for the **trace alignment** problem, which can be reduced to the problem of finding a counterfactual via automata path-search.

4.5.1 Trace Alignment

Trace alignment is a **conformance checking** technique that involves minimally editing a sequence of events—called a **trace log**—by adding or removing events to ensure conformity with a given **process model**.

As stated in Adriansyah et al. (2011):

> Conformance checking measures how good a model of a process is with respect to an event log that records the executions of the process.

The work of De Giacomo et al. (2017) demonstrates that if the process is modeled using a LTL_f formula, the trace alignment problem can be solved through a planning algorithm over automata. The trace alignment problem is formally defined as:

> The trace alignment problem is defined as follows: given a trace t and an LTL_f formula ϕ s.t. $t \not\models \phi$, find a trace t' s.t. $t' \models \phi$ and $\text{cost}(t, t')$ is minimal.

In this approach, two automata are introduced: a **trace automaton** $\mathcal{T}$ and a **constraint automaton** $\mathcal{A}$.

The **trace automaton** is a deterministic automaton constructed to accept only the trace t. It is formally defined as:

$$\mathcal{T} = \langle \Sigma_t, Q_t, q_0^t, \rho_t, F_t \rangle$$

where:
21. $\Sigma_t = \{e_1, \ldots, e_n\}$ is the set of n events in t.

1. $Q_t = \{q_0^t, \ldots, q_n^t\}$ is a set of $n+1$ arbitrary states.
2. $\rho_t = \bigcup_{i=0}^{n-1} \langle q_i^t, e_{i+1}, q_{i+1}^t \rangle$ is the set of transitions defining the progression from one state to the next upon reading an event.
3. $F_t = \{q_n^t\}$ is the final state.

Conversely, the **constraint automaton** is non-deterministic and is constructed to accept only traces that satisfy the formula ϕ. It is formally defined as:

$$\mathcal{A} = \langle \Sigma, Q, q_0, \rho, F \rangle$$

where:
25. $\Sigma = \{e_1, \ldots, e_m\}$ is the set of m possible events (the event universe).

1. $Q = \{q_0, \ldots, q_n\}$ is a set of $n+1$ arbitrary states.
2. $\rho = \bigcup_{i=0}^{n-1} \langle q_i, e_{i+1}, q_{i+1} \rangle$ is the set of transitions defining the progression between states upon reading an event.
3. $F \subset Q$ is the set of accepting states.

To adapt both $\mathcal{T}$ and $\mathcal{A}$ for the trace alignment problem, the authors propose a series of **augmentations**.

From $\mathcal{T}$, they derive an **augmented automaton**:

$$\mathcal{T}^+ = \langle \Sigma_t^+, Q_t, q_0^t, \rho_t^+, F_t \rangle$$

where:

- Σ_t^+ includes all propositions in Σ_t, along with a **deletion proposition** del_p for every $p \in \Sigma$, and an **addition proposition** add_p for every $p \in \Sigma \cup \Sigma_t$.

- ρ_t^+ includes all transitions in ρ_t, plus:
  - A new transition $\langle q, del_p, q' \rangle$ for every transition $\langle q, p, q' \rangle \in \rho_t$.
  - A new transition $\langle q, add_p, q \rangle$ for every proposition $p \in \Sigma_t$ and state $q \in Q_t$, provided there is no existing transition $\langle q, p, q' \rangle \in \rho_t$ for any $q' \in Q_t$.

The new **repair propositions** (add_p and del_p) allow $\mathcal{T}^+$ to accept any trace over Σ that results from repairing t, with repairs explicitly marked by these propositions.

For example, given a trace $t = ab$:

- $\mathcal{T}^+$ will accept both t and the modified trace $t' = del_a b$.
- However, it will **not** accept $t'' = b$ because the deletion of a is not explicitly marked.

From $\mathcal{A}$, we derive an augmented automaton:

$$\mathcal{A}^+ = \langle \Sigma^+, Q, q_0, \rho^+, F \rangle$$

where:

- $\Sigma^+ = \Sigma_t^+$ (matching the augmented event set of $\mathcal{T}^+$).
- ρ^+ includes all transitions in ρ, plus:
  - A new transition $\langle q, del_p, q \rangle$ for all $q \in Q$ and $p \in \Sigma_t$.
  - A new transition $\langle q, add_p, q' \rangle$ for every transition $\langle q, \psi, q' \rangle \in \rho$ such that $p \models \psi$.

For example, consider a trace $t = a, b$ and a constraint automaton $\mathcal{A}$ for the formula $\varphi = \Box(b \rightarrow \Diamond c)$.
Since $t \not\models \varphi$ (because c does not occur after b), t is **not** accepted by $\mathcal{A}$ nor by $\mathcal{A}^+$.

However, if we repair t by adding c at the end, the new trace $\hat{t} = a, b, add_c$ will be accepted by $\mathcal{A}^+$ because now $\hat{t} \models \varphi$.

Now it's possible to notice that:

- $\mathcal{T}^+$ accepts all traces obtained by applying repair propositions to the original trace.
- $\mathcal{A}^+$ accepts only those traces that contain repair propositions **and** satisfy the LTL_f formula.

By taking the **synchronous product** of $\mathcal{T}^+$ and $\mathcal{A}^+$—which represents the simultaneous execution of the same input on both automata—we obtain a new automaton that accepts only those traces that result from applying repair propositions to the original trace and that satisfy the formula.

This automaton serves as the **domain for a cost-optimal planning problem**, which seeks to find the minimal set of edits required to align the trace with the process model.

4.5.2 Deterministic Finte Automaton as a Local Surrogate Model

Taking inspiration from the constraint automaton proposed by De Giacomo et al. (2017), we define a **deterministic** finite automaton (DFA) $\mathcal{A}_{\text{condition}}$. This automaton processes sequences x_i of characters from a universe of characters $\mathcal{I}$ and **accepts** if and only if a given *condition* holds; otherwise, it rejects.

Formally, the DFA is defined as:

$$\mathcal{A}_{\text{condition}} = \langle \Sigma, Q, q_0, \rho, F \rangle$$

where:
29. $\Sigma = \{i_1, \ldots, i_n\} = \mathcal{I}$ represents the alphabet of n characters.

1. $Q = \{q_0, \ldots, q_m\}$ is a set of $m + 1$ states, where q_0 is the initial state.
2. $\rho = \bigcup_{i=0}^{m-1} \langle q_i, i_{i+1}, q_{i+1} \rangle$ defines the state transitions when reading a character.
3. $F \subset Q$ is the set of final (accepting) states.

Given:

- A source point x,
- A label $y = \mathcal{F}(x)$ produced by an oracle model $\mathcal{F}$ (whose internals are unknown),
- A target label $\hat{y}$,

we define two distinct automata:

- **Targeted Automaton** ($\mathcal{A}_t$): Accepts sequences x_i only if $\mathcal{F}(x_i) = \hat{y}$—i.e. the label is equal to the target label.
- **Non-Targeted Automaton** ($\mathcal{A}$): Accepts sequences x_i only if $\mathcal{F}(x_i) \neq y$—i.e. the label is different from the ground truth label.

The distinction between the **categorized** and **uncategorized** settings is unnecessary in the context of the automaton, as the automaton's logic remains unchanged regardless of whether a label represents a category or an individual item.

For complex models $\mathcal{F}$ (e.g., deep neural networks), constructing an exact DFA that generalizes to every possible x_i is impractical. Instead, we can build a **local surrogate DFA**, which approximates the behavior of the black-box model $\mathcal{F}$ **within the neighborhood** of the source point x.

Intuitively, the automaton $\mathcal{A}$ represents the decision-making process of $\mathcal{F}$ for each point in a sequence.

Example: Suppose $\mathcal{A}$ is the automaton depicted in Figure 4.3, and we have a sequence $x_1 = [1, 4, 3, 2]$. Running this sequence through $\mathcal{A}$ results in an accepted final state, meaning the condition is satisfied ($\mathcal{F}(x_1) = \hat{y}$ if $\mathcal{A}$ is targeted, $\mathcal{F}(x_1) \neq y$ if not). Additionally, truncating x_1 to the first two characters results in the sequence $[1, 4]$, which still leads to an accepting final state. This indicates that the truncated sequence also satisfies the target label condition.

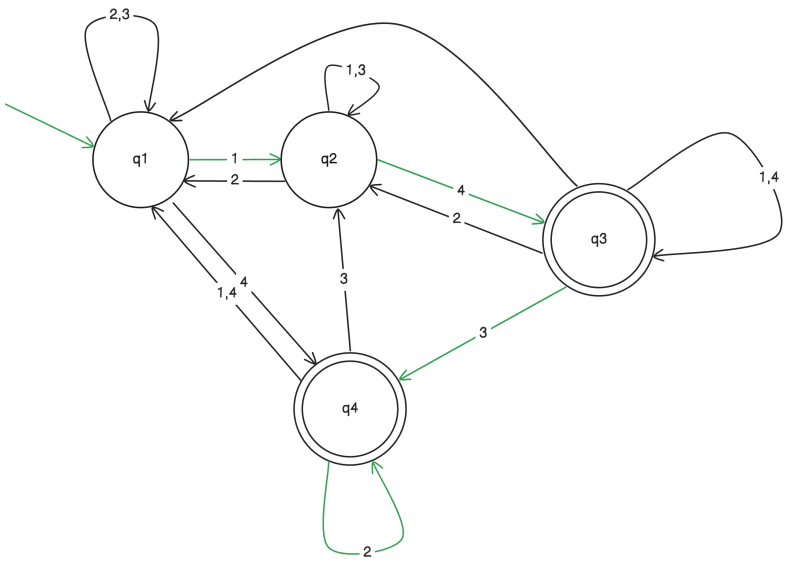


Figure 4.3. An example of a possible automaton $\mathcal{A}$

Including possible edits in the DFA

If we want to check the condition for an **edited version** of x_1, a straightforward approach would be to first modify x_1, obtaining a new sequence x_1', and then execute each character of x_1' on $\mathcal{A}$ to verify the condition.

However, this approach is inefficient because it does not leverage the **transparency** of the automaton, making it no better than directly evaluating the edited sequence on the black-box model $\mathcal{F}$.

Consider a source sequence $x = [1, 2, 4]$ and assume we want to test the condition over all possible **single-edit** variations of x, meaning sequences that maintain an edit distance of 1 from x. Since the closer a modified sequence is to x, the higher the chance of it being correctly evaluated by the automaton, we focus on minimal modifications.

Given an alphabet of characters $\mathcal{I} = \{1, 2, 3, 4, 5\}$, the possible sequences with one modification are:

- $x_1 = [3, 2, 4]$, $x_2 = [5, 2, 4]$,
- $x_3 = [1, 3, 4]$, $x_4 = [1, 5, 4]$,
- $x_5 = [1, 2, 3]$, $x_6 = [1, 2, 5]$.

In this case, the automaton needs to read 25 characters to test all variations. This approach does not exploit the automaton's structure and could be replaced by testing all sequences directly on the black-box model $\mathcal{F}$.

To efficiently incorporate edits into the automaton itself, we use the **repair propositions**

proposed in De Giacomo et al. (2017). We extend the automaton's alphabet with two additional special symbols: add_c and del_c, where c is the character modified by the proposition. These symbols represent actions applied to the sequence. For example, given a source sequence $x = [1, 2, 4]$, the modified sequence:

$$x_i = [1, \text{add}_3, 2, \text{del}_4]$$

is equivalent to:

$$x_i = [1, 3, 2]$$

since add_3 inserts 3, and del_4 removes 4.

We now define the **augmented automaton** $\mathcal{A}^+_{\text{condition}}$ as:

$$\mathcal{A}^+_{\text{condition}} = \langle \Sigma^+, Q, q_0, \rho^+, F \rangle$$

where:

1. Σ^+ contains all characters in Σ plus two new symbols: del_p and add_p for all $p \in \Sigma$.
2. ρ^+ contains:
   - All transitions in ρ.
   - A new transition $\langle q, \text{del}_p, q \rangle$ for all $q \in Q$ and $p \in \Sigma$, ensuring that deleting a character results in a self-loop.
   - A new transition $\langle q, \text{add}_p, q' \rangle$ for all $\langle q, p, q' \rangle \in \rho$, ensuring that adding a character follows the same transition as reading it.

The intuition Behind ρ^+ is that:

- If we are in state $q \in Q$ due to the sequence $(p_1, p_2, p_3) \in \Sigma$ and want to **delete** the next character $p_4 \in \Sigma$, instead of executing p_4 (which may lead to a different state q'), we execute del_{p_4}, which loops on q, acting like the character was never in the sequence to begin with.
- If we are in state $q \in Q$ and want to **add** a new character $p_5 \in \Sigma$ that was not in x, executing add_{p_5} transitions exactly as reading p_5 would.

This distinction allows us to differentiate between:

- A **synchronous** action (sync_p), where we read a character present in the sequence.
- An **edit** action (add_p or del_p), where we insert a new character that was not originally in the sequence or delete a character that was present in the sequence.

4.5.3 Differences from the work by De Giacomo et al. (2017)

While our automata construction is heavily inspired by the approach used for the **trace alignment problem** in De Giacomo et al. (2017), our setting introduces key differences, leading to several modifications.

In the original work, the authors employed the **synchronous product** of a **non-deterministic** augmented constraint automaton $\mathcal{A}^+$ and a **deterministic** augmented trace automaton $\mathcal{T}^+$. This combination was used to accept sequences that were generated from the source sequence using repair propositions and satisfied a given formula ϕ.

In contrast, our approach only uses the **deterministic** augmented constraint automaton $\mathcal{A}^+$, where the formula ϕ is defined based on the **condition** and the **black-box model**. Specifically, a sequence is accepted by the formula only if the output of the black-box model aligns with the condition. Here, coherence is defined in terms of **similarity to the target label or the source label**.

In the original work, the **trace automaton** was used to ensure that the execution of the automaton followed the original source sequence when searching for a solution. In our case, this constraint is enforced **at runtime** using only $\mathcal{A}^+$, which eliminates the need for an explicit synchronous product. This is crucial because computing such a product would result in an **exponential explosion** in the number of transitions and states.

Another fundamental difference is that in De Giacomo et al. (2017), the **constraint automaton $\mathcal{A}^+$ is non-deterministic (NFA)** because it is designed to enforce **temporal logic constraints** expressed as a **LTL$_f$ formula**. Since **Linear Temporal Logic (LTL)** defines conditions that may hold over multiple paths (i.e., different possible future sequences), the automaton needs **non-determinism** to explore multiple possible ways of satisfying the formula. In particular, the NFA representation allows multiple transitions from the same state on the same input character, enabling the automaton to **"guess"** different paths that might satisfy the formula ϕ. Since the formula may require satisfying **temporal constraints** (e.g., an event must happen at some point in the future), the automaton needs to keep track of multiple possibilities simultaneously. This non-determinism is resolved by synchronizing the execution of the trace automaton and constraint automaton through their **synchronous product**, ensuring that only feasible sequences are explored.

In contrast, in our approach, we use a **deterministic (DFA) version of $\mathcal{A}^+$** because instead of enforcing a temporal logic formula, our condition is a **boolean predicate** on the output of the black-box model $\mathcal{F}$ (e.g., "does $\mathcal{F}(x)$ produce label $\hat{y}$?"). This means that **each sequence is either accepted or rejected immediately**, without the need for backtracking or tracking multiple possibilities. Because our automaton **directly encodes the decision boundary of $\mathcal{F}$**, each input uniquely determines the next state, making non-determinism unnecessary.

4.5.4 Automata learning

In this section, we describe the construction of the automaton $\mathcal{A}_{\text{condition}}$.

Given a dataset $\mathcal{D}$ consisting of points labeled as either $True$ or $False$, we can learn a deterministic finite automaton (DFA) that accepts points with the $True$ label and rejects those with the $False$ label. If $\mathcal{D}$ is generated based on a condition—where "good" points (labeled $True$) satisfy the condition and "bad" points (labeled $False$) do not—the resulting learned automaton will accept only points that meet this condition.

The dataset $\mathcal{D}$ is generated using GENE, ensuring both positive (satisfying) and negative (non-satisfying) examples. The condition itself is defined as $\tilde{y} = y_t$—if targeted, with y_t being the target—or $\tilde{y} \neq y$—if untargeted, with y being the source sequence's label. When learning automata, there are two main approaches: **active learning** and **passive learning**.

Active learning approach This method relies on an **oracle model**, a black-box system that acts as a teacher by answering queries. The goal is to minimize the number of queries while maximizing learning efficiency. One of the most well-known active learning algorithms is $L^\star$, introduced by Angluin (1987), which learns **regular languages** as DFAs through interaction with the oracle. The algorithm uses two types of queries:

- **Membership queries (MQs)**: Used to check whether a given string belongs to the target language.
- **Equivalence queries (EQs)**: Used to verify whether a hypothesized DFA correctly represents the target language.

Passive learning approach Unlike active learning, passive learning relies on a **static dataset**, without access to an oracle for querying. A key algorithm in this domain is **RPNI** (Regular Positive and Negative Inference), introduced by Oncina and García (1993).

RPNI learns a DFA from a finite set of positive and negative examples, corresponding to the dataset $\mathcal{D}$. The algorithm starts with a **prefix tree acceptor (PTA)** that represents the positive examples and incrementally merges states while ensuring consistency with the negative examples. It ultimately produces a minimal DFA that aligns with the given data.

4.5.5 Constraint A*: Counterfactual generation as a path-search problem

This section details the algorithm that, given the augmented automaton $\mathcal{A}^+$ constructed from a source sequence x, generates a counterfactual sequence by traversing the shortest path from the initial state to a final state.

Let $q_0 \in Q$ be the initial state of the augmented automaton $\mathcal{A}^+$. The algorithm assigns a cost of 1 to add_p and del_p operations, represented by the edge weights, while all other transitions have a cost of 0, simulating what we call a sync_p action. If a sequence contains a (add_p, del'_p) or (del'_p, add_p) combination, the total cost is 1 instead of 2, since this operation effectively replaces p' with p, maintaining an edit distance of 1 from the original sequence.

Given these edge weights, traversing the shortest path from q_0 to any final state $q \in F$ results in a path $(e_1, e_2, \ldots, e_n)$ with a cumulative cost c. This path represents the sequence of actions $e_i \in \Sigma^+$ required to transform x into its counterfactual version $\tilde{x}$, with exactly c edits.

To find this path, we must ensure that for each character in x, an explicit action (add, del, or sync) is taken. The algorithm cannot terminate before an action has been executed for every character in x, even if it reaches an accepting state.

We propose the **Constrained** $A^\star$ algorithm, a modified version of the classic path-search $A^\star$ algorithm. This adaptation allows it to operate on the automaton while enforcing the constraint that each character in the input sequence must be processed through an explicit action.

Algorithm Overview

The main part of the algorithm is the `AStar` function (pseudocode in Algorithm 1), which uses a modified version of A* search to explore possible edits by considering both the cost of the current edit and a heuristic estimate of the remaining cost to reach a target state in the DFA. The `getNeighbours` function (pseudocode in Algorithm 2) is used to generate valid transitions for the current state while enforcing the constraints on the edits.

The `AStar` function takes several parameters: the **DFA** that defines the states and transitions, the **current state** from which the search starts, the **target states** (which are the accepting states in the DFA), and the **remaining part** of the trace that needs to be aligned. It also takes a **set of constraints** on the allowable alignment length, a **heuristic function** for estimating the cost to the nearest target state, and any **initial alignment actions**. The initial and remaining alignment are useful when we want to constrain the algorithm in only editing a subsequence.

The algorithm begins by initializing a **priority queue** (called `paths`), which stores potential solution paths. Each entry in the queue consists of a tuple that includes the **current cost** of the path, the **heuristic value** (which estimates the remaining cost to reach a target state), and other information like the **current state**, the **sequence of actions** taken so far, and the **index** in the **remaining trace** that still needs to be processed.

With the initial path setup, the algorithm enters the search loop. At each step, it pops the path with the lowest total cost (current cost plus heuristic) from the priority queue. For each path, the algorithm generates potential neighbors using the `getNeighbours` function, which checks which state transitions are valid, based on the current trace character and the constraints.

For each valid neighbor, the algorithm calculates the **new cost** and creates a **new path** that extends the current path with this neighbor. If the neighbor represents a valid alignment that reaches one of the target states and satisfies the length constraints, the algorithm **immediately returns** the corresponding sequence of edit actions, that when applied to the source sequence will result in a counterfactual.

The `getNeigbhour` function plays a crucial role in the search by generating valid transitions for the current state. It ensures that only legal actions are considered during the search and that the constraints on subsequence editing are respected.

When generating neighbors, the function first retrieves the **possible transitions** from the current state in the DFA. For each transition, it checks if the action is **legal** based on the

current state, **current read char** and **action type** (whether it is a synchronization, deletion, or addition), making sure the action is consistent with the rest of the process. For example, if an *add* action has already chosen for the item 5, then further *sync* actions cannot be done on the same character, otherwise the sequence will result in duplicate characters. Or if a character has not been added or synched yet, it cannot be deleted.

The function then evaluates the cost of each action. For synchronization actions (denoted by SYNC), the cost is **zero** if the current character in the trace matches the transition element. For deletion (DEL) and addition (ADD) actions, the cost is **one**, but if the previous action was a corresponding opposite action (such as a delete followed by an add), the cost is reduced to **zero**, as the combination of both actually represent a character **replace**, which causes a single character to be edited instead of 2.

Once the valid neighbors are identified, the function returns them to the `AStar` function, which continues the search by adding these neighbors to the priority queue.

Algorithm 1 Constrained A*

Require: Deterministic finite automaton $\mathcal{A}^+$, origin state s_0, target states T, source sequence $x = (x_1, x_2, \ldots, x_n)$
Ensure: Optimal alignment sequence or `None` if no valid alignment exists
Initialize priority queue queue $\leftarrow \emptyset$
Add tuple $(0, 0, 0, s_0, [s_0], (), n)$ to queue
Initialize empty dictionary visited $\leftarrow \emptyset$
while queue is not empty **do**
Pop (cost, heuristicValue, counter, currentState, path, actions, remainingTraceIdx) from queue
if remainingTraceIdx $= 0$ **and** currentState $\in T$ **then return** actions
end if
currentChar $\leftarrow x[|x| -$ remainingTraceIdx$]$ **if** remainingTraceIdx > 0 **else** `None`
neighbors $\leftarrow$ getNeighbors($\mathcal{A}^+$, currentState, currentChar)
for all (targetState, actionCost, action) $\in$ neighbors **do**
if (currentState, currentChar, action) $\in$ visited **then**
continue
end if
newCost $\leftarrow$ cost + actionCost
newPath $\leftarrow$ path + [targetState]
newActions $\leftarrow$ actions + [action]
newRemainingTraceIdx $\leftarrow$ remainingTraceIdx $- 1$ **if** action modifies the trace **else** remainingTraceIdx
heuristicValue $\leftarrow$ *heuristic*(targetState, newActions)
Add (newCost, heuristicValue, counter + 1, targetState, newPath, newActions, newRemainingTraceIdx) to queue
Add (currentState, currentChar, action) to visited
end for
end while
return `None`

Algorithm 2 Get Valid Neighbors

Require: Deterministic finite automaton $\mathcal{A}$, current state s, current character c
Ensure: List of neighbors (targetState, actionCost, action)

```
Initialize empty list neighbors ← ∅
Initialize set of candidate states candidateStates ← set of transitions of s
Initialize set of inputs inputsSet ← set of available actions in DFA
for all action ∈ candidateStates do
    (actionType, e) ← decodeAction(action)
    targetState ← transition of s for action
    if (actionType, e) is legal with respect to inputsSet then
        if actionType = SYNC and c = e then
            Add (targetState, 0, action) to neighbors
        else if actionType = DEL and c = e then
            cost ← 1
            if last action in inputs is ADD then
                cost ← 0
            end if
            Add (targetState, cost, action) to neighbors
        else if actionType = ADD then
            cost ← 1
            if last action in inputs is DEL then
                cost ← 0
            end if
            Add (targetState, cost, action) to neighbors
        end if
    end if
end for
    return neighbors
```

Edit a subsequence

To restrict the algorithm to editing only a specific subsequence $x_{(i:j)}$, where i is the starting index and j is the final index, we divide the sequence into three parts:

1. **Executed sequence**: $x_{(0:i-1)}$, representing the prefix that remains unchanged.
2. **Mutable sequence**: $x_{(i:j)}$, the portion of the sequence where edits are allowed.
3. **Fixed sequence**: $x_{(j+1:\text{end})}$, representing the suffix that must remain unchanged.

We set the initial state q_0 of the automaton $\mathcal{A}^+$ as the state reached after processing the executed sequence. The set of target states T is defined as the states that, after processing the fixed sequence, lead the automaton to an accepting state $q \in Q$.

The **Constrained** $A^\star$ algorithm ensures that only the **mutable** sequence is modified when generating a counterfactual. It does so by executing the algorithm with starting state q_0 and target states T, effectively limiting modifications to $x_{(i:j)}$.

Additionally, the algorithm must consider character uniqueness constraints. Since the **executed** and **fixed** sequences must not contain duplicates, if a character p_1 is already present in these sequences, an add_{p_1} action is prohibited to prevent duplication.

Example of execution

In this section, we will showcase an example of the **Constrained** $A^\star$ algorithm execution on a toy source sequence and automaton.

The source sequence is $x = [1, 3, 2]$, while the universe of items is $\mathcal{U} = \{1, 2, 3, 4, 5\}$. The automaton learned after generating the dataset of good and bad points is shown in Figure 4.4.

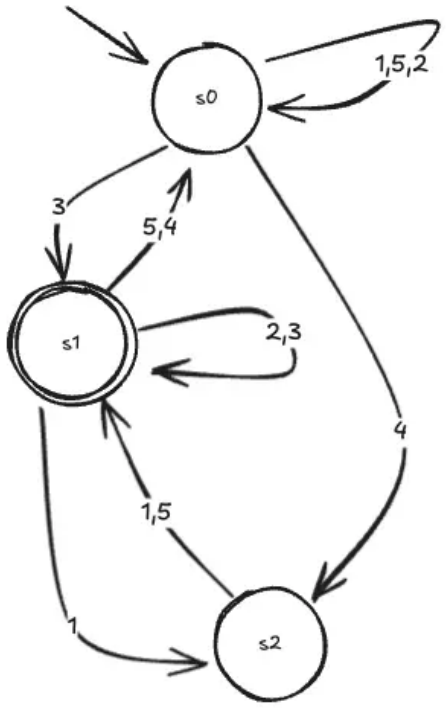


Figure 4.4. The learned automaton $\mathcal{A}$ using RPNI.

This automaton accepts the sequence x and all other sequences in the **good** dataset (points similar to x and with the same label), while rejecting the sequences in the **bad** dataset (points that differ from x and have a different label, i.e., counterfactuals).

We augment the automaton with the rules described earlier, adding the add_c and del_c actions, now depicted in Figure 4.5. Since we want to find the sequence that transitions from being accepted (good point) to non-accepted (counterfactual), we simplify the algorithm by inverting the automaton (Figure 4.6). Now, we only need to find a sequence that will be accepted (counterfactual).

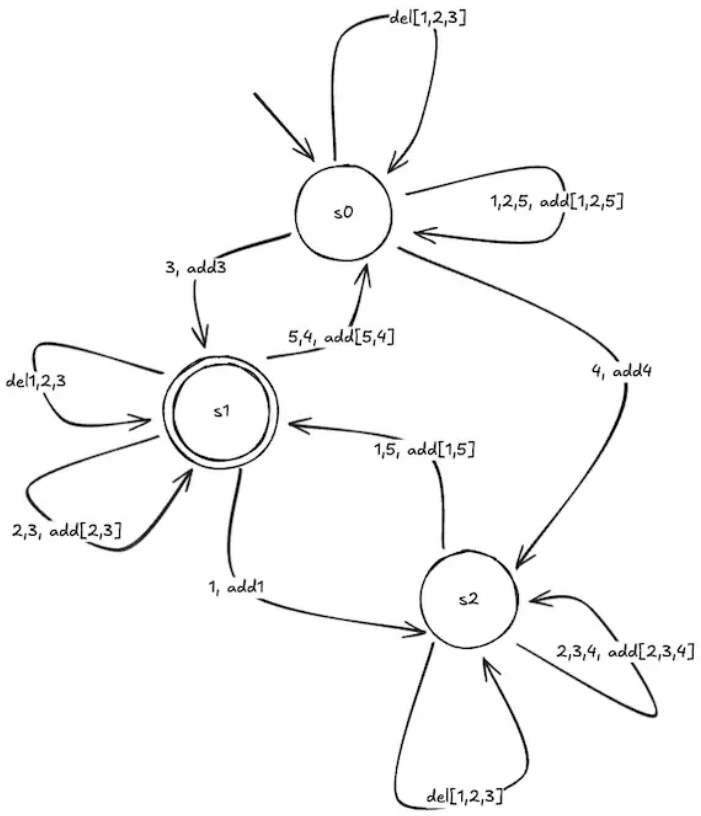


Figure 4.5. The augmented automaton $\mathcal{A}^+$ with the repair propositions.

Given $x = [1, 3, 2]$ and the initial state $q_0 = s_0$, we execute the **Constrained Dijkstra** algorithm (Dijkstra instead of $A^\star$ is used here for simplicity, as we are ignoring the heuristic function).

1. Initially, $i = 0$, $x_0 = 1$, and the current state is $q = s_0$. The possible actions we can perform are $\{\text{sync}_1, \text{del}_1, \text{add}_2, \text{add}_3, \text{add}_4, \text{add}_5\}$. The model performs the sync_1 action. After traversing the edge with label 1, the state remains s_0.

2. The next character is $x_1 = 3$. The possible actions are $\{\text{sync}_3, \text{del}_3, \text{add}_2, \text{add}_4, \text{add}_5\}$. The model performs sync_3. The current state is now s_1, and the list of actions is $[\text{sync}_1, \text{sync}_3]$.

3. The next character is $x_2 = 2$. The possible actions are $\{\text{sync}_2, \text{del}_2, \text{add}_4, \text{add}_5\}$. The model chooses sync_2. The current state remains s_1, and the list of actions is $[\text{sync}_1, \text{sync}_3, \text{sync}_2]$. There are no characters left to read from x, but the current state is not a target state, so the algorithm continues.

4. The next character is `null`. The possible actions are $\{\text{add}_4, \text{add}_5\}$. Both actions lead to accepting states, so the model chooses one of them. Let's say the model chooses

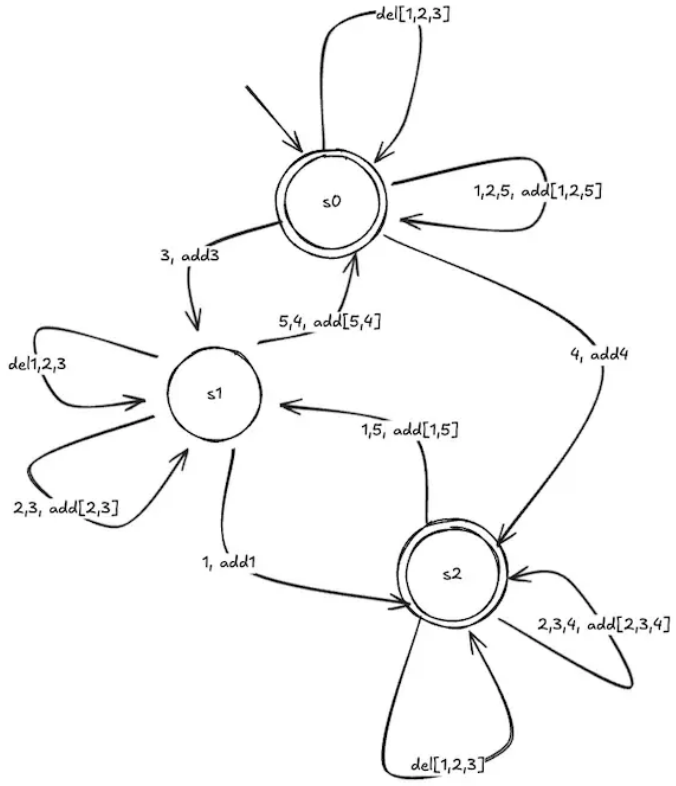


Figure 4.6. The inverted augmented automaton

add_4. The state is now s_0, it is accepting, and there are no remaining characters to read from x. Therefore, the model returns the sequence of actions (alignment).

5. The sequence of actions $[\text{sync}_1, \text{sync}_3, \text{sync}_2, \text{add}_4]$ is converted into the counterfactual sequence $\tilde{x} = [1, 3, 2, 4]$. The counterfactual sequence has an edit distance of 1 from the original source sequence.

Sometimes, we may want to constrain the resulting counterfactual to have a certain length. In the previous example, we've seen how the method produces an explanation that can be converted into: "If the user had interacted with item 4, the recommendation would have been different (or equal to the target, depending on the setting)." While this is a valid explanation, in some cases, the user may prefer an explanation that does not simply add interactions (increasing the sequence length), but rather replaces some interactions, keeping the sequence length the same.

To achieve this, we can use the two parameters from the algorithm description: `minAlignmentLength` and `maxAlignmentLength`. By setting both parameters to the length of the source sequence, we can ensure that the counterfactual sequence has the same length as the source sequence.

This forces the algorithm to continue execution until the result is within the specified length range, even if all other conditions are met.

To clarify this process, let's revisit the same example as before, but with the addition of alignment length conditions (`minAlignmentLength = 3` and `maxAlignmentLength = 3`).

1. Step 1 is the same as before.
2. Step 2 is the same as before.
3. The next character is $x_2 = 2$. The possible actions are $\{\text{sync}_2, \text{del}_2, \text{add}_4, \text{add}_5\}$. The model first chooses sync_2, but since the next actions can only be add_4 or add_5, the path will be discarded because it cannot result in a sequence of length 3. The model discards sync_2 and all other add actions, as they would produce alignments longer than 3. The only remaining option is the action del_2. The current state is now s_1, and the list of actions is $[\text{sync}_1, \text{sync}_3, \text{del}_2]$. There are no characters left to read from x, but the current state is not a target state, and the alignment has length 2, which is less than 3. Therefore, the algorithm must continue.
4. Step 4 is the same as before.
5. The sequence of actions $[\text{sync}_1, \text{sync}_3, \text{del}_2, \text{add}_4]$ is converted into the counterfactual sequence $\tilde{x} = [1, 3, 4]$.

As we can see, the constraint on the sequence length has resulted in a counterfactual with a cost of 2 but with the same length as the original sequence. The explanation can be phrased as: "If the user had interacted with item 4 instead of item 2, the recommendation would have been different (or equal to the target, depending on the setting)."

Heuristic

A **heuristic** is a function that estimates the value of a given decision in an optimization problem, aiming to guide the algorithm toward an optimal or near-optimal solution more efficiently.

While the heuristic is not a mandatory component of the algorithm, it can significantly reduce the execution time in certain cases by directing the search toward promising paths, avoiding unnecessary exploration of less optimal solutions.

In path-search algorithms, heuristics often represent the estimated "distance" to the target. For example, in scenarios where the vertices of the graph correspond to geographic coordinates, the **Euclidean distance** between the current state and the target state can serve as an effective heuristic, helping the algorithm prioritize paths leading toward the goal.

In our case, however, the graph vertices do not correspond to geographic locations or coordinates. Instead, the states of the automaton represent different conditions or transitions in a sequence. Nonetheless, we still possess key insights into the problem structure that can inform the heuristic design. Specifically, we know that `sync` actions have no cost, and therefore, an optimal alignment will involve maximizing the number of `sync` actions

while minimizing the number of `add` and `del` operations to ensure the automaton reaches a target state.

Using these insights, we can design a heuristic function that provides an estimate of the remaining work in the search process:

1. **Start with the current character** x_i in the trace and the remaining part of the trace, denoted as $x_{(i+1:)}$.
2. Attempt to execute as many `sync` actions as possible for each character x_j in the remaining trace $x_{(i+1:)}$ while ensuring that the transition for each character exists. For each $x_j \in x_{(i+1:)}$, try to execute the sync_{x_j} action until no more sync_{x_j} actions can be performed, either because there is no transition for that character or because we have reached the end of the trace.
3. Once the synchronization actions are exhausted, the automaton will be in a new state q_i'.
4. **Perform a breadth-first search (BFS)** from the new state q_i' to all possible target states. This step finds the minimum number of transitions (hops) required to reach one of the target states.
5. The **heuristic value** is then the hop distance from the new state q_i' to the nearest target state.

Unlike standard $A^\star$ algorithms, where the heuristic value is added to the current cost to form the total evaluation function, in our case, the heuristic is used **only as a tie-breaker** when two paths have the same cost. This means that if two paths have the same total cost, the one with the lower heuristic value will be prioritized, as it is expected to bring the automaton closer to a target state in fewer transitions.

4.5.6 PACE Pipeline

To summarize, the PACE pipeline (Figure 4.7) to find a counterfactual starting from a source sequence x is the following:

1. Use the genetic algorithm defined in the GENE method to produce a dataset of good and bad points, each of those point close to x;
2. Feed the dataset to RPNI to learn a DFA $\mathcal{A}$;
3. Augment $\mathcal{A}$ with repair propositions to allow edits, obtaining $\mathcal{A}^+$;
4. Execute the constraint $A^\star$ algorithm on $\mathcal{A}^+$ to get the minimal sequence of edits that transforms the source point to a counterfactual point;
5. Execute those edits on x to obtain the counterfactual $\tilde{x}$.

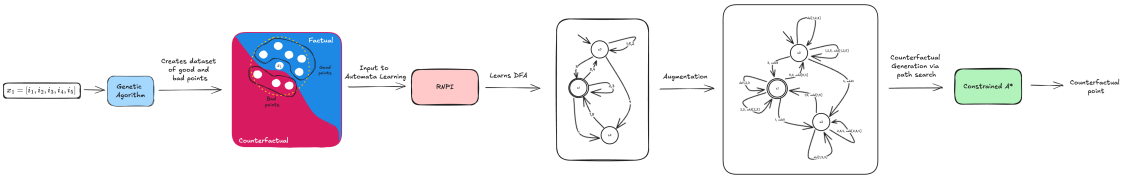


Figure 4.7. Diagram showing the pipeline of PACE.

4.6 Computational Complexity & Scalability

4.6.1 Brute-force vs. heuristic approaches

To analyze the complexity of the optimization problem, we can look at the complexity of an algorithm that solves the problem using **brute force**, generating all the possible sequences $\tilde{\mathcal{H}}$ of length n and then individuate one of the optimal counterfactuals.

Note that the optimality of a solution doesn't imply its uniqueness, since if more sequences are valid counterfactuals and have the same exact distance from the source sequence, they all are considered optimal counterfactuals.

The time complexity of the brute force approach is shown in the work of Guidotti (2024) for feature vectors, hence we can easily adapt it to our setting.

Let's define as m the number of items in the universe, and n the length of the sequence we want to find the counterfactual of, then the brute force time complexity is:

$$O\left(\binom{m}{n} \cdot m' \cdot r\right)$$

Where m' is the maximum number of items to vary simultaneously and r is the maximum number of values to test for each modified item. In our case, since each item can be replaced with each other item, r is equal to m.

Note that with this case, we are limiting the solution space to the counterfactual sequences of the same length as the original sequence. If we want to find counterfactuals with a range of lengths from n_0 to n, then the complexity will be:

$$O\left(\sum_{n_i=n_0}^{n} \binom{m}{n_i} \cdot m' \cdot r\right)$$

Since the complexity is not polynomial, the problem is NP-hard.

Targeted counterfactual generation is harder

We can see how the **targeted** problem is *harder* than the **untargeted** problem by analyzing the number of valid solutions in the search space we just defined.

Let V_u be the set of valid counterfactual sequences in the **untargeted** setting; V_t the set of valid counterfactual sequences in the **targeted** setting, and S the entire search space (the set of all possible counterfactual sequences).

In the **untargeted setting**, we are seeking any sequence that is a valid counterfactual, i.e., a sequence that differs from the original in some way. The set of valid solutions V_u is larger because we are not constrained by any specific target. The only requirement is that the counterfactual deviates from the original sequence, but there are no further restrictions on what that counterfactual should achieve.

In the **targeted setting**, the counterfactual must not only deviate from the original sequence, but it must also meet a specific target or condition. This target can be a desired outcome, such as achieving a specific classification or meeting a particular performance threshold. Due to this additional constraint, the number of valid counterfactual solutions V_t is smaller than in the untargeted case, because the counterfactual sequence must satisfy this extra requirement.

Since the valid solutions V_t are constrained by the target, it follows that:

$$|V_t| < |V_u|$$

The ratio of valid solutions to the entire search space S is smaller in the targeted setting. In the untargeted setting, we are simply searching for valid counterfactuals, so the ratio $\frac{|V_u|}{S}$ is larger. However, in the targeted setting, since there are fewer valid solutions, the ratio $\frac{|V_t|}{S}$ is smaller.

Hence, we have:

$$\frac{|V_t|}{S} < \frac{|V_u|}{S}$$

4.6.2 Suboptimal solution with brute force

Since **sequential recommenders** are particularly sensitive to modifications in the **last elements** of a sequence, we can use this to our advantage by designing a **suboptimal** brute-force approach with a significantly lower time complexity.

Instead of modifying the entire sequence, we limit changes to only the **last** k elements of the original sequence, where $k \ll m$. This reduces the number of possible modifications, leading to a new time complexity:

$$O\left(\binom{m}{k} \cdot m' \cdot r\right)$$

If we **also** allow the counterfactual sequence to have a different length, the complexity generalizes to:

$$O\left(\sum_{n_i=n_0}^{k} \binom{m}{n_i} \cdot m' \cdot r\right)$$

where n_i represents the allowed sequence lengths, ranging from n_0 to k.

This approach is **suboptimal** because it does **not** explore the entire solution space. By restricting modifications to only the last k elements, we **reduce** the number of possible counterfactuals we can generate, potentially missing more optimal solutions.

However, this trade-off allows us to **significantly reduce** computational complexity while still finding meaningful counterfactuals.

To further **accelerate** the brute-force search, we can introduce an **early stopping** mechanism, meaning that instead of exploring all possible modifications, we **stop** once we find a valid counterfactual that meets a predefined **distance threshold** from the original sequence, avoiding exhaustively searching all possibilities if we found a "good enough" solution.

For very small values of k (e.g., $k = 2$) and a limited number of simultaneous modifications m' (e.g., 1 or 2), this brute-force approach remains computationally feasible, **provided that** r is also small. However, since the growth remains **exponential in** k, the approach quickly becomes infeasible as k increases.

Chapter 5

Implementation Details

To implement the system discussed in this thesis, several frameworks and libraries were used, each one playing a crucial role in making a specific problem much easier to deal with. This section provides a discussion on the libraries used and how they have been implemented inside the project. [1]

5.1 Open Source Libraries

5.1.1 Recbole

Citing its GitHub [2]:

> RecBole is developed based on Python and PyTorch for reproducing and developing recommendation algorithms in a unified, comprehensive and efficient framework for research purpose. Our library includes 91 recommendation algorithms.

RecBole implements the three sequential recommendation models we intended to evaluate: GRU4Rec, SASRec, and BERT4Rec.

- **GRU4Rec** (Hidasi et al., 2016), (Hidasi and Karatzoglou, 2018): employs **Gated Recurrent Units (GRUs)** to model sequential user interactions, allowing it to capture both **short-term** and **long-term** dependencies, reducing the vanishing gradient problem standard RNNs suffer from. By incorporating ranking-aware loss functions such as **BPR** (Bayesian Personalized Ranking) and **TOP1**, and later introducing **TOP1-max**, it optimizes ranking performance in recommendation tasks. This architecture significantly improved upon traditional matrix factorization approaches, demonstrating better adaptability to session-based recommendation scenarios.

[1] The codebase is hosted publicly on GitHub at https://github.com/DomizianoScarcelli/counterfactual-explanations-recsys.

[2] RecBole repository https://github.com/RUCAIBox/RecBole

- **SASRec** (Kang and McAuley, 2018): introduced a **self-attention mechanism** for sequential recommendation, moving away from recurrent architectures. Inspired by the success of Transformers in NLP, it captures long-range dependencies while selectively focusing on the most relevant past interactions. The use of **multi-head self-attention** makes it computationally efficient, allowing it to learn richer user representations while being significantly faster than RNN-based models. In particular, the self-attention mechanism solves the two key limitation of earlier methods: the need for large amounts of dense data; and the struggle in capturing complex dynamics in user behavior.
- **BERT4Rec** (Sun et al., 2019): further evolved the self-attention paradigm by incorporating **bidirectional encoding**, unlike SASRec's left-to-right processing. Using a **Masked Language Model (MLM) objective**, it learns user preferences by predicting randomly masked items within an interaction sequence. This allows BERT4Rec to leverage both past and future interactions, achieving state-of-the-art performance in multiple benchmark datasets, such as MovieLens 1M and 20M, Steam and Amazon Beauty.

RecBole also includes utilities to train models on various datasets and facilitates dataset handling through its `Interaction` object, a Python dictionary containing key information about data points, such as:

- `user_id`: a `torch.Tensor` of shape `[batch_size]` representing the unique identifier of the user involved in the interaction.
- `item_id_list`: a `torch.Tensor` of shape `[batch_size, MAX_LENGTH]` representing the sequence of item IDs that the user with `user_id` has interacted with, in chronological order. This represents the user's interaction history, and it's the main piece of data used to recommend the next item.
- `item_length`: the length of the user's interaction history sequence. This field is essential since all the tensors in `item_id_list` are of length `MAX_LENGTH`, but they are zero-padded, meaning that we should only consider the first `item_length` characters.
- `rating_list`: a tensor of shape `[batch_size, MAX_LENGTH]` containing the sequences of ratings or implicit feedback corresponding to each item in `item_id_list`. Each rating indicates how the user interacted with the respective item.
- `timestamp_list`: similarly to the `timestamp` field, it's a tensor of shape `[batch_size, MAX_LENGTH]` containing the sequence of timestamps for the interactions in `item_id_list`. Each timestamp indicates when each interaction in the interaction sequence happened.

During the training phase, we will have further fields, all `torch.Tensor` with shape `[batch_size]`:

- `item_id`: representing the ground truth of the element that should be recommended next, available only in the training split.
- `rating`: representing the rating (or implicit feedback) the user with `user_id` has given to the item with `item_id`. It could be an explicit score (e.g., 4.0 for a 5-star rating) or an implicit signal (e.g., 1 for interaction, 0 for no interaction).

- `timestamp`: representing the timestamp of the interaction the user had with `item_id` in seconds since the epoch in the Unix format.

For models like **BERT4Rec**, the `Interaction` object also includes details about mask tokens, such as their positions in the sequence.

Using RecBole to run a genetic algorithm on a chosen dataset while making inferences on a model—and later hot-swapping datasets and models for performance evaluation—was significantly easier than building everything from scratch with PyTorch alone. Additionally, using a well-established library like RecBole ensures thoroughly tested code, reducing the likelihood of introducing bugs.

5.1.2 AALpy

According to its GitHub page [3], *AALpy* is a lightweight **automata learning** library written in Python. It provides classes to model various types of automata, such as DFAs, Moore, and Mealy machines, along with **active and passive automata learning algorithms** for learning these models.

In this project, *AALpy* was used to efficiently learn a DFA using the RPNI passive learning algorithm, to augment the learned DFA, and to run sequences on it. Without this library, it would have been necessary to manually implement a DFA class with all its mechanisms, along with a function to execute the RPNI passive learning algorithm.

5.1.3 DEAP

According to its GitHub page [4], **DEAP** is a novel evolutionary computation framework designed for rapid prototyping and testing of ideas.

In this project, DEAP was used to streamline the creation of functions for dataset generation using a genetic algorithm. The framework's core features were adapted as follows:

- DEAP's creator module defined the structure of individuals and their fitness function, which was designed to minimize costs.
- The `toolbox` module registered the genetic operations:
- `Individuals` were initialized from the source sequence.
- The initial population consisted of copies of the source sequence.
- The `mate` function was configured with DEAP's built-in two-point crossover function.
- The `mutate` function was linked to custom mutations. This function randomly applied mutations to a percentage of the population.
- The `select` function utilized DEAP's tournament selection strategy.

[3] AALpy repository https://github.com/DES-Lab/AALpy
[4] DEAP repository https://github.com/DEAP/deap

Additionally, the evolutionary algorithm was customized to optimize fitness evaluation by performing recommender model inference in batches of individuals, enhancing efficiency.

5.2 Evaluation Setup

We conducted the evaluation using two versions of the MovieLens dataset: **100K** and **1M**. First released in 1998, the MovieLens dataset captures user preferences for movies in the form of <user, item, rating, timestamp> tuples, where ratings range from **0 to 5** (Harper and Konstan, 2016).

As one of the most widely used datasets for recommender systems, MovieLens has multiple versions that vary in size, including **100K, 1M, 10M, and 20M**, where the number in the name corresponds to the total number of interactions (tuples) in the dataset. In this work we used the 100K and 1M versions, which details are showcased in Table 5.1.

Name	Date Range	Rating Scale	Users	Movies	Ratings	Density
ML 100K	9/1997 - 4/1998	1-5, stars	943	1,682	100,000	6.30%
ML 1M	4/2000 - 2/2003	1-5, stars	6,040	3,706	1,000,209	4.47%

Table 5.1. Summary statistics for the used versions of the MovieLens dataset.

Train, validation, test split strategy When dealing with *sequential recommendation*, the dataset is used to construct sequences of interactions, each representing a user. This is achieved by grouping interactions by user and chronologically sorting them using the timestamp field.

When training and evaluating a sequential recommender model, the standard random train/validation/test splitting technique is unsuitable because it does not preserve the temporal order of interactions.

For example, consider a user's interactions: $[(i_1, t_1), (i_2, t_2), (i_3, t_3), (i_4, t_4), (i_5, t_5)]$ where i represents an interaction, and t is the timestamp such that: $t_1 < t_2 < t_3 < t_4 < t_5$.

A random split might produce:

- **Train set:** $[(i_1, t_1), (i_3, t_3), (i_5, t_5)]$
- **Validation set:** $[(i_4, t_4)]$
- **Test set:** $[(i_2, t_2)]$

This disrupts the temporal sequence, an essential property for ensuring that the model predicts future interactions based only on past data.

A better approach is the **leave-one-out** split strategy, which preserves temporal order by reserving the last interaction for the test set and the second-to-last for validation. This

results in:

- **Train set:** $[(i_1, t_1), (i_2, t_2), (i_3, t_3)]$
- **Validation set:** $[(i_4, t_4)]$
- **Test set:** $[(i_5, t_5)]$

This method ensures that the model learns only from past interactions when predicting the next interaction.

Using RecBole, we trained all three previously mentioned models for **300 epochs** on both variations of the MovieLens dataset. We primarily used the default hyperparameter values provided by the library (BERT4Rec [5], SASRec [6], GRU4Rec [7]).

Training was performed with **negative sampling** using a **uniform** distribution, with the number of negative samples set to **100**.

Table 5.2 presents the evaluation of these models using standard recommender system metrics after training.

Model	Dataset	Recall@10	NDCG@10	Precision@10
GRU4Rec	ml-100k	0.6235	0.3444	0.0624
	ml-1m	0.5015	0.2764	0.0501
SASRec	ml-100k	0.6585	0.3880	0.0659
	ml-1m	0.7980	0.5786	0.0798
BERT4Rec	ml-100k	0.6755	0.4119	0.0676
	ml-1m	0.7829	0.5416	0.0783

Table 5.2. Evaluation metrics of the used models

5.2.1 Reducing search space with target popularity

When using the **targeted** approach, evaluating all possible targets can be computationally expensive. In the case of the MovieLens dataset, there are 18 different movie genres, and running the evaluation on each of them would require a significant amount of time. However, intuitively, the method should yield similar performance for targets that share a comparable level of popularity. Therefore, instead of evaluating all genres, we can focus on a subset of targets that act as representatives of their respective popularity ranges.

[5] BERT4Rec default hyperparameters: https://recbole.io/docs/user_guide/model/sequential/bert4rec.html

[6] SASRec default hyperparameters: https://recbole.io/docs/user_guide/model/sequential/sasrec.html

[7] SASRec default hyperparameters: https://recbole.io/docs/user_guide/model/sequential/gru4rec.html

MovieLens 100K

For the MovieLens 100K dataset, the **category** popularity was computed to help determine which targets should be selected. The computed popularity is presented in Table 5.3, where the **Frequency** column represents the proportion of items in the dataset that belong to a given category (also depicted in Figure 5.1); while the **Interaction-Level Frequency** column indicates how often items of a given category were interacted with by users (also depicted in Figure 5.2).

Class	Frequency	Percentage (%)	Interaction-Level Frequency	Interaction-Level Percentage (%)
Drama	725	43.08	39,895	18.77
Comedy	505	30.01	29,832	14.03
Action	251	14.91	25,589	12.04
Thriller	251	14.91	21,872	10.29
Romance	247	14.68	19,461	9.15
Adventure	135	8.02	13,753	6.47
Children's	122	7.25	7,182	3.38
Crime	109	6.48	8,055	3.79
Sci-Fi	101	6.00	12,730	5.99
Horror	92	5.47	5,317	2.50
War	71	4.22	9,398	4.42
Mystery	61	3.62	5,245	2.47
Musical	56	3.33	4,954	2.33
Documentary	50	2.97	758	0.36
Animation	42	2.50	3,605	1.70
Western	27	1.60	1,854	0.87
Film-Noir	24	1.43	1,733	0.82
Fantasy	22	1.31	1,352	0.64
unknown	2	0.12	10	0.00

Table 5.3. Category popularity in the MovieLens 100K dataset.

By analyzing these values, we can identify a handful of categories that span different popularity levels, allowing us to perform evaluations more efficiently while maintaining representativeness across the dataset.

The categories chosen for evaluation, listed in order of popularity, are: **Drama**, **Action**, **Adventure**, **Horror**, and **Fantasy**.

To ensure that hyperparameter tuning could be completed within a reasonable time frame, we limited the evaluation to the **Action** category. Action has a balanced level of popularity, making it a suitable representative for other categories. Once the optimal hyperparameters were determined, the full evaluation was conducted across the selected representative targets.

In the **uncategorized** setting, only *interaction-level frequency* is considered (depicted in

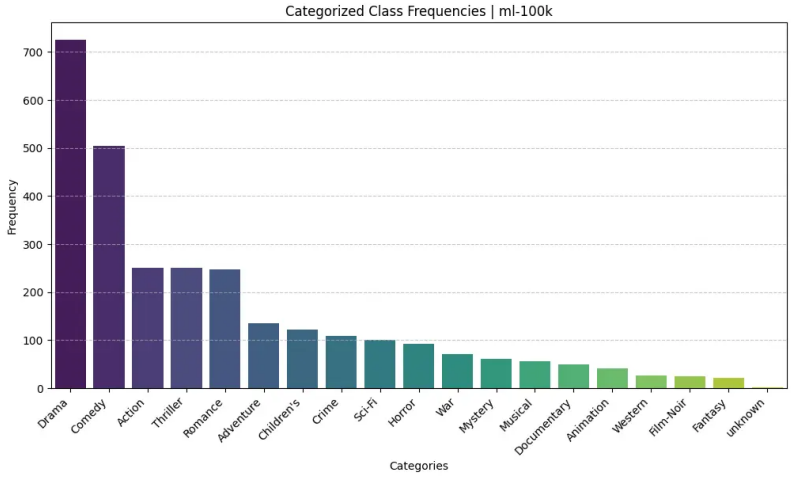


Figure 5.1. Categorized class frequencies (MovieLens 100K)

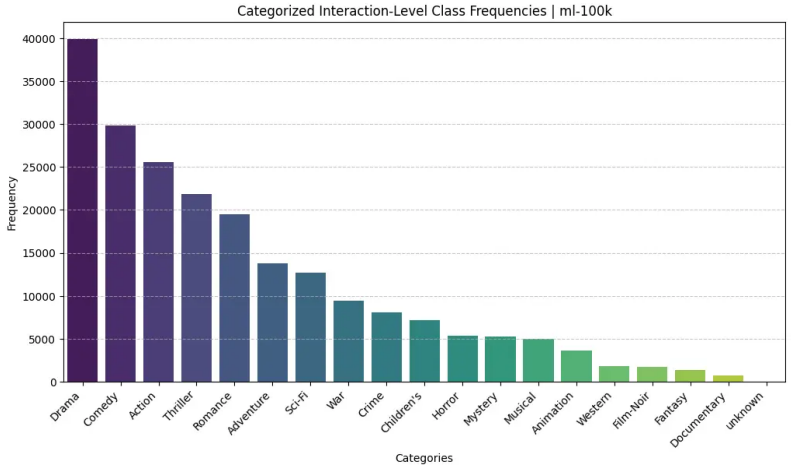


Figure 5.2. Categorized interaction-level class frequencies (MovieLens 100K)

Figure 5.3), as each item is treated as its own category—indicated with its ID number in the figure. This means that standard frequency values are always 1 for every item. Instead, popularity is computed based on the number of times an item has been interacted with by users.

The plot at Figure 5.3 illustrates a classic example of a *long-tail distribution*, where the majority of items have very few interactions, while a small subset of items are significantly more popular.

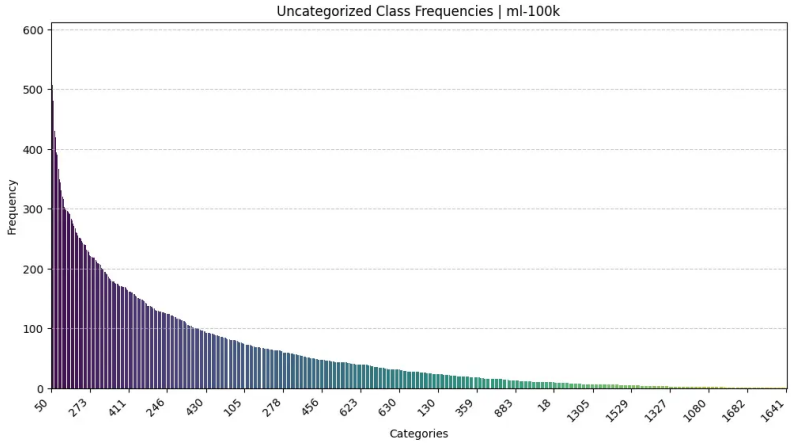


Figure 5.3. Uncategorized class frequencies (MovieLens 100K)

MovieLens 1M

While for the MovieLens 1M dataset, the histogram of frequencies for all the settings are shown in Figure 5.4, Figure 5.5 and Figure 5.6.

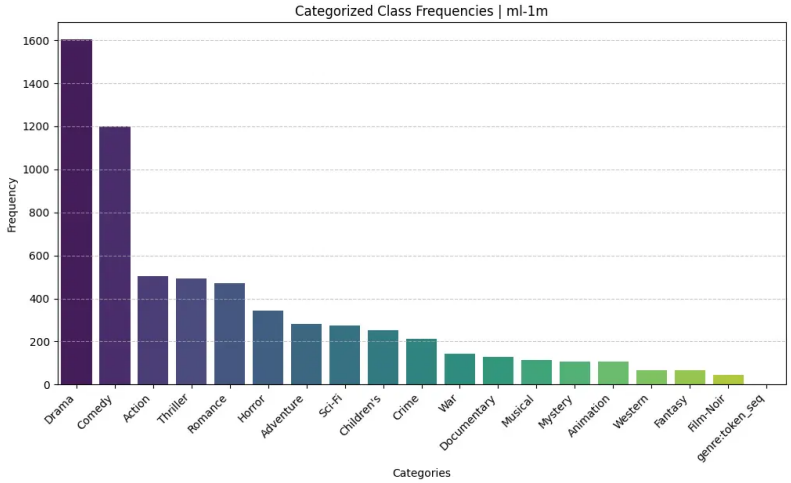


Figure 5.4. Categorized class frequencies (MovieLens 1M)

For what regards the category class frequencies, the proportion is very similar to the proportion in the 100K version of the dataset.

5.2.2 Evaluation result store

To accelerate the evaluation of all models, users, and settings, we ran multiple combinations concurrently across different scripts and machines. To efficiently manage concurrent writes,

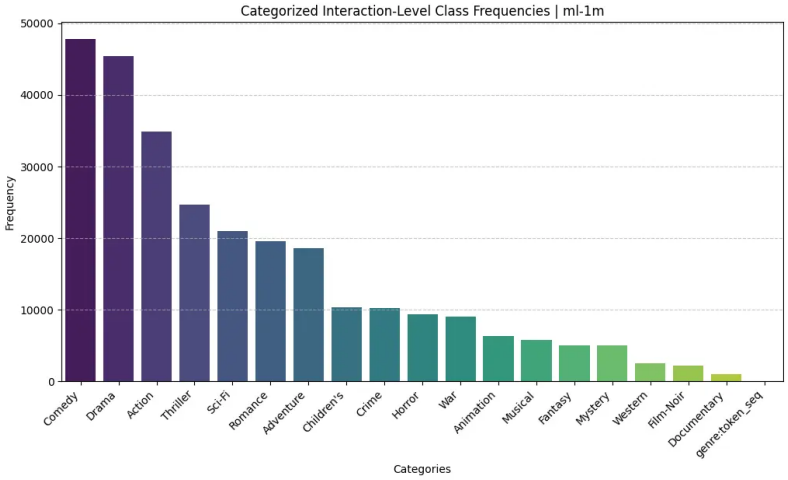


Figure 5.5. Categorized interaction-level class frequencies (MovieLens 1M)

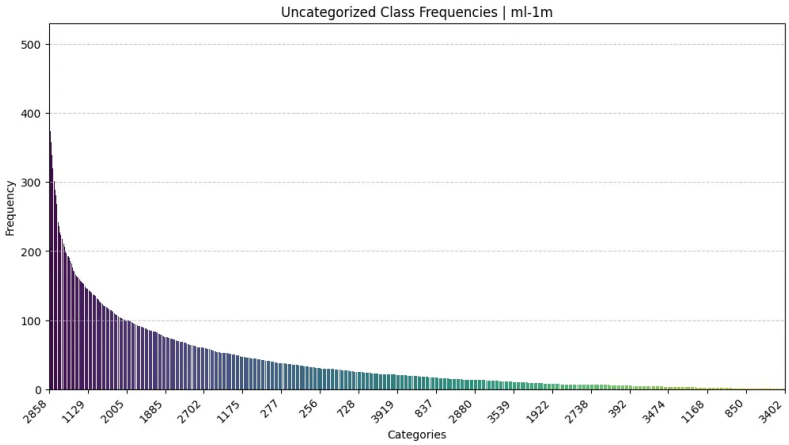


Figure 5.6. Uncategorized class frequencies (MovieLens 1M)

we stored each user's counterfactual evaluation in an **SQLite** database.

5.2.3 Action encoding

In the constraint A* algorithm, each edge is labeled with an **action**, which is composed by its **action type** (i.e., SYNC, ADD or DEL) and the **action subject**—i.e., the character on which the action is being performed. In order for the algorithm to not perform the same action given the same current character that is being read and current state, we need to store this information inside a set. Storing actions like strings is not optimal, for two main reasons: strings are not memory efficient; and the two pieces of information (type and subject) would be merged together into a string, meaning that inspecting this information would require further processing (string splitting and integer conversion).

To address both of these issues, we adopt **action encoding**, where actions are stored as raw bits. By doing this, we achieve **memory efficiency** and ensure that **actions are easy to inspect and compare** without additional string processing.

Each action is encoded as a **15-bit integer**, where:

- The **first 2 bits** represent the **action type** (SYNC, ADD, or DEL).
- The **remaining 13 bits** represent the **action subject** (the character being modified). This value would have to be modified in order for the number of bits to be able to encode each item in the dataset.

Chapter 6

Evaluation

6.1 Model Sensitivity

An experiment conducted during the development of the project aimed to assess how sensitive sequential recommender models are to changes in user interaction sequences. The goal was to quantify the impact of **single-item changes** in a sequence, measured as the NDCG between the labels associated with the original and modified sequences.

Given the universe of items $\mathcal{I}$, the procedure is the following:

1. For each user interaction sequence x of length m and position i, create a matrix M of shape $(m, |\mathcal{I}| - 1)$. Each row $M[j]$ corresponds to a sequence identical to x apart from the item at position i, which is the replaced with the j-th item from the universe of items $\mathcal{I}$. There are a total of $\mathcal{I} - 1$ sequences because the original sequence x is excluded.

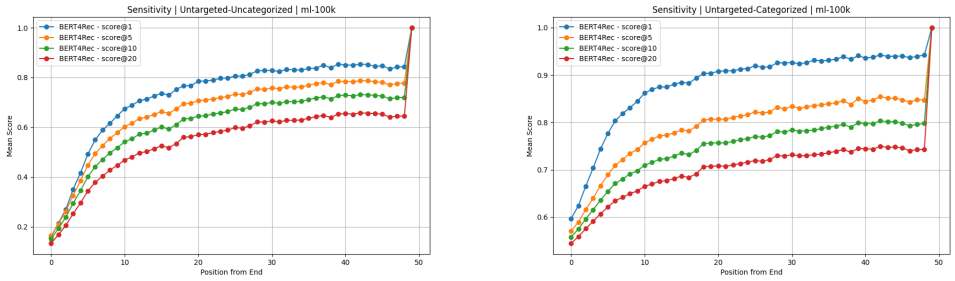
$$M = \begin{bmatrix} x_1, \ldots, i_1, \ldots, x_m \\ x_1, \ldots, i_2, \ldots, x_m \\ x_1, \ldots, i_3, \ldots, x_m \\ \vdots \\ x_1, \ldots, i_{|\mathcal{I}|-1}, \ldots, x_m \end{bmatrix}$$

   where i_j is the j-th item from $\mathcal{I}$.

2. Pass the matrix M to the sequential recommender model. The model produces a matrix of logits M_o with shape $|\mathcal{I}| \times |\mathcal{I}| - 1$, where each row corresponds to the model's output logits (shape $\mathcal{I}$) for each of the modified sequences.

3. Pass the raw logits to a post-process function $\mathcal{G}(M_o, k)$ which processes the output according to the setting, takes the top-k recommendation and outputs the NDCG similarity score according to the Counterfactual@k framework.

4. The previous step produces $|\mathcal{I}| - 1$ scores. The mean of those scores will be the sensitivity metric for the configuration source sequence and edit position.



(a) Uncategorized (b) Categorized

Figure 6.1. Model Sensitivity for **untargeted** case

The experiment was conducted on the BERT4Rec model using the MovieLens 100K dataset. From the plots in Figure 6.1, it is possible to observe how the NDCG@k score varies between the original and modified sequences when a single edit is made at a specific position. When modifying an element in the last positions (represented by lower numbers in the plot, as positions are counted from the end of the sequence to account for varying sequence lengths), the NDCG@k score approaches 0, indicating a complete shift in recommendations from the original ones. Conversely, the NDCG@k value increases when the edit occurs near the middle or the beginning of the sequence. This trend is observed in both categorized and uncategorized cases, though in the categorized case, the change is less pronounced, with the NDCG@k stabilizing around the 0.5–0.6 range in the worst-case scenario. It is important to note that when an element appears in the first position of the array (position 49 from the end inside the plots), the NDCG@K remains 1.0 for all tested sequences. This suggests that the model is not sensitive to changes in the first element of the sequence.

In conclusion, the **model sensitivity** experiment demonstrates that sequential recommender models exhibit high sensitivity to single-item changes in the last positions. This makes counterfactual identification a trivial task in the uncategorized case and an almost-trivial task in the categorized case. To address this, we extended our evaluation by testing the proposed counterfactual generation methods using a targeted approach. This adjustment increases the difficulty of the task by restricting the number of valid solutions within the solution space.

6.2 Counterfactual Generation Evaluation

In this section, we discuss the steps used to evaluate counterfactual generation with both **GENE** and **PACE**.

The evaluation serves multiple purposes: **finding the best hyperparameters** based on evaluation metrics, **assessing the performance** of the methods, and determining which method is **more effective** for counterfactual generation.

6.2.1 Evaluation Metrics

The chosen evaluation metric is **model fidelity**, which measures how well the method explains results. Specifically, it represents the percentage of valid counterfactuals over the total number of generated explanations. Since counterfactual validity depends on both the similarity threshold and the parameter k (which determines the number of items considered for similarity), model fidelity is computed for different values of k (i.e., fidelity@1, fidelity@5, fidelity@10, and fidelity@20). The similarity threshold is treated as a hyperparameter, as it influences the distribution of good and bad points in the dataset generated by the genetic algorithm. To assess how close the generated counterfactual is to the source sequence, we used the **edit distance**.

The computation of the ground truth—i.e., the label y on which the counterfactual's label $\hat{y}$ is compared to—is different depending on the setting.

- If the setting is **untargeted**, the ground truth is computed by taking the *argmax* of the raw logits, which are the output of the forward pass to the black-box model given the source sequence x. The argmax operation retrieves the indices of the top-k best-recommended items, ordered by confidence level: $[i_1, i_2, \ldots, i_k]$. If the setting is **uncategorized**, this ranking of items directly represents the ground truth. However, if the setting is **categorized**, each item is mapped to its corresponding set of categories, resulting in a list of category sets encoded as integers: $[\{c_1, c_2\}, \{c_3, c_4\}, \ldots, \{c_5, c_6\}]$.
- If the setting is **targeted** and **categorized**, then the **ground truth** is a list of k sets of the target category (encoded as integer): $[\{\text{target}\}, \{\text{target}\}, \ldots, \{\text{target}\}]$. If **uncategorized**, then we just interpret each different item as its own category. Differently from the categorized approach, we take into account the fact that each category (item) can appear only once in each ranking.

A recap on how the ground truth is computed depending on the setting is depicted in Table 6.1.

Setting	**Ground Truth**
Untargeted & Uncategorized	Top-k items: $[i_1, i_2, \ldots, i_k]$
Untargeted & Categorized	Top-k item categories: $[\{c_1, c_2\}, \{c_3, c_4\}, \ldots, \{c_5, c_6\}]$
Targeted & Categorized	Target category: [{target_cat}, {target_cat}, ..., {target_cat}]
Targeted & Uncategorized	Target item: [{target_item}, {target_item}, ..., {target_item}]

Table 6.1. Ground truth computation across different settings.

After computing the ground truth, we follow the same process for counterfactual sequences. Given raw logits $\hat{y}$ for a counterfactual sequence $\tilde{x}$, we extract the prediction array using the recommender model and compute similarity using NDCG@k. This produces a score between 0 and 1.

Given a top-k extraction and the corresponding set of scores S_k, along with threshold t,

model fidelity is computed as:

$$\text{Model Fidelity@k} = \frac{1}{|S_k|} \sum_{i=1}^{m} \mathbb{1}(S_k[i] \geq t),$$

where $\mathbb{1}$ is an indicator function that returns 1 if the condition is met, otherwise 0.

If the genetic algorithm fails to find a counterfactual, the "bad" portion of the dataset will be empty, causing an exception. Since the automata cannot be learned on an empty dataset, the process must be halted. In such cases, a score of 0 (counterfactual not found) is added to the list of scores for both methods.

To assess the distance between the counterfactual and the original sequences, we employ edit distance—the same metric used in the genetic algorithm's fitness function. To ensure that only valid counterfactuals are considered, we introduce the concept of **Distance@k**. This metric represents the edit distance between the source label x and a counterfactual label $\tilde{x}$, where the counterfactual is considered valid at k—that is, its NDCG@k exceeds the similarity threshold (or falls below the similarity threshold in the case of the targeted approach). Since a single sequence may be valid counterfactual at one k value, but not valid at another k' value, it will be involved in the Distance@k but not in the Distance@k'.

6.2.2 GENE Evaluation

The following hyperparameters are used when generating the dataset with the genetic algorithm:

- `similarity_threshold (float)`: A decimal number between 0 and 1 representing the threshold for similarity between the outcomes of two sequences (computed with NDCG@k). If the similarity score is above this threshold, the outcomes are considered equal. Unlike standard cases where thresholding is done post-hoc, here it is used during dataset generation to distinguish between "good" and "bad" points, leading to different datasets based on the chosen threshold.
- `ignore_genetic_split (bool)`: The *Constrained* $A^\star$ algorithm can take an optional *sequence split* input, defining the subsequence to be edited. If `false`, the genetic algorithm also respects this split, mutating only the *mutable* portion of the sequence. If `true`, the entire sequence is mutated.
- `topk (List[int])`: A list of k values used to compute the NDCG@k score between the counterfactual logits and the source sequence logits after generating a counterfactual sequence.
- `genetic_topk (int)`: The k value used in the genetic algorithm to compute the NDCG@k score for determining whether a point belongs in the "good" dataset. It has to be noted that while in the `topk` parameter, multiple k values can be used, here only one can be used since it will determine the dataset split, hence different k values will yield different dataset splits.
- `generations (int)`: The number of generations the genetic algorithm runs.

- `pop_size (int)`: The number of individuals (population size) in each generation of the genetic algorithm.
- `target (str)`: Used only in the **targeted** setting, representing the target value as a string (category or item).
- `halloffame_ratio (float)`: The ratio of top-performing individuals retained across generations.
- `fitness_alpha (float)`: A weighting factor that balances the distance between the source sequence and the current sequence against the distance between the source label and the current label when computing fitness in dataset generation.
- `allowed_mutations (List[str])`: A list of allowed mutations, which must be a subset of all possible mutations.
- `mut_prob (float)`: The probability that an individual undergoes mutation.
- `crossover_prob (float)`: The probability that an individual undergoes crossover with another individual.
- `num_additions (int)`: The number of **addition** mutations performed when an individual is selected for the **addition** operation.
- `num_deletions (int)`: The number of **deletion** mutations performed when an individual is selected for the **deletion** operation.
- `num_replaces (int)`: The number of **replace** mutations performed when an individual is selected for the **replace** operation.
- `targeted (bool)`: Indicates whether counterfactual generation is targeted, meaning the algorithm aims to generate counterfactuals that match `target`.
- `categorized (bool)`: Indicates whether counterfactual generation is categorized, meaning labels are represented by their categories.

6.2.3 PACE Evaluation

Constrained A* Algorithm Hyperparameters

In addition to the genetic algorithm hyperparameters, generating a counterfactual using the *constrained* $A^\star$ algorithm over the automata requires defining the following hyperparameters:

- `include_sink (bool)`: During RPNI, a **sink state** is added to ensure the automaton is input-complete. Since this state is non-accepting, it becomes accepting when the automaton is inverted (as per the counterfactual generation algorithm). This parameter controls whether the sink state should be considered a target state.
- `split (tuple)`: This parameter defines how the input sequence is divided, allowing the model to modify only a specific portion of the original sequence. The tuple consists of three elements: the length of the **executed** part (which determines the

starting state of the automaton before the constrained $A^\star$ execution), the length of the **mutable** part (which can be altered to generate the counterfactual), and the length of the **fixed** part (which remains unchanged at the end). For example, consider the tuple `(None, 5, 10)`. Here, the last 10 elements of the sequence are fixed, meaning they cannot be modified. The 5 elements before them from the mutable part, which the model is allowed to edit. The remaining portion at the beginning of the sequence is the executed part, which is used to determine the automaton's starting state. If the sequence length is 50, this tuple translates to `(35, 5, 10)`, meaning the first 35 elements define the initial state of the automaton, the next 5 elements are subject to modification, and the last 10 elements remain unchanged. A more flexible configuration is `(None, 10, None)`, where both the executed and fixed parts are unspecified. If the sequence length is 50, the remaining elements are split equally, resulting in `(20, 10, 20)`. This ensures that the split rule adapts dynamically to different sequence lengths. The `None` values in the tuple allow for defining a **split rule** that remains independent of sequence length, which is particularly useful when dealing with large datasets. Since executing the path-search algorithm on an entire sequence is computationally expensive, two commonly used splits are `(None, 10, 0)` and `(None, 5, 0)`. These settings constrain the model to modify only the last 10 or 5 elements of the sequence, respectively, reducing computational complexity while maintaining flexibility in counterfactual generation.

6.2.4 Hyperparameter tuning

To efficiently **evaluate** counterfactual generation using both **GENE** and **PACE**, a streamlined approach was used. At the end of the dataset generation process, the sequence in the "bad" dataset that is closest to the source sequence is selected as the **best counterfactual**. This sequence is then used to compute and log the evaluation metrics for GENE. Following this, the "good" and "bad" datasets are used to train the automaton, and the counterfactual generated by PACE is evaluated in the same way. This approach allows **both methods** to be assessed using a single PACE generation, optimizing efficiency.

Hyperparameter tuning via **grid search** was conducted across various parameter combinations.

```
pop_size = [2048, 4096, 8192, 16384]
generations = [10, 20, 30]
fitness_alpha = [0.5, 0.7, 0.85]
similarity_threshold = [0.5, 0.7]
mut_prob = [0.2, 0.5]
crossover_prob = [0.2, 0.5]
split = [(None, 5, 0), (None, 10, 0)]
genetic_topk = [1, 5, 10]
num_additions = [1,2,3]
num_deletions = [1,2,3]
num_replaces = [1,2,3]
```

Since a full hyperparameter search is computationally expensive, some configurations were manually discarded early if they performed worse on the first (n) evaluated se-

quences compared to previously tested configurations. This **early pruning** significantly reduced execution time while ensuring that only promising configurations were explored further.

The **best-performing configuration** identified through hyperparameter tuning is as follows:

```
topk = [1, 5, 10, 20]
include_sink = true
similarity_threshold = 0.5
ignore_genetic_split = true
genetic_topk = 1
generations = 10
pop_size = 8192
halloffame_ratio = 0
fitness_alpha = 0.5
allowed_mutations = [
    "replace",
    "swap",
    "add",
    "delete",
    ]
mut_prob = 0.5
crossover_prob = 0.7
num_additions = 1
num_deletions = 1
num_replaces = 1
```

One notable aspect of the chosen configuration is that `ignore_genetic_split` is set to `true`. This means that although the `split` parameter is configured as `(None, 10, 0)`, which restricts Constrained A* to modifying only the last 10 elements of a sequence, the genetic algorithm disregards this restriction and is allowed to edit any part of the source sequence.

At first glance, this may seem to disadvantage Constrained A*, as it limits how many items can be changed, potentially reducing model fidelity. However, evaluations of GENE with `ignore_genetic_split` set to `false` showed no difference in performance. The explanation for this, supported by model sensitivity experiments, is that the last 5 to 10 elements of a sequence tend to have the most significant impact on the final decision. Therefore, if a valid counterfactual exists, the explainer should be able to find it by modifying only those elements.

Despite similar performance, the decision to keep `ignore_genetic_split` as `true` was made because allowing edits at different positions in the sequence produces a more diverse dataset for automata learning. This, in turn, may improve the robustness of the automaton, leading to better generalization in counterfactual generation.

6.2.5 Evaluation results

The evaluation was conducted on three different models: **BERT4Rec**, **SASRec**, and **GRU4Rec**, using two datasets: **MovieLens 100K** and **MovieLens 1M**. For MovieLens 100K, GENE and PACE were run on **all 943 users**. In contrast, for MovieLens 1M, the evaluation was performed on a **subset of 200 users**.

From the computed model fidelity, we observe that **more frequent targets**—both in the categorized (Table 6.2) and uncategorized (Table 6.3) settings—achieve **higher fidelity** across both proposed methods, GENE and PACE. In general, GENE attains **near-perfect fidelity@1** in almost all settings—summarized in Table 6.2 for **targeted-categorized**, Table 6.4 for **untargeted-categorized**, and in Table 6.5 for **untargeted-uncategorized**—indicating that it successfully identifies at least one valid counterfactual for nearly all sequences in both the MovieLens 100K and MovieLens 1M datasets.

The **targeted-uncategorized** task presents the greatest challenge due to the sparsity of the solution space. For highly popular targets in the **MovieLens 100K** dataset, such as the item with ID 50, GENE achieves **relatively high fidelity**, explaining over 50% of the data points. However, for **less frequent** targets like the item with ID 1305, both methods struggle to explain most data points. As intuitively expected, **GENE consistently outperforms PACE**. This is because GENE generates the dataset on which PACE's automaton is trained, and while the automaton requires numerous data points to train effectively, GENE only needs a single valid point to achieve high fidelity. For what regards the **targeted-uncategorized** evaluation on the **MovieLens 1M** dataset, the performances are highly worse, with most of the counterfactual generation processes failing to generate a dataset, which causes both GENE and PACE to yield a fidelity of 0.

This phenomenon doesn't happen for the other settings, where the bigger MovieLens 1M dataset seem to provide with better performances.

As expected, fidelity@k generally decreases as k increases, since longer rankings are inherently more difficult to match. However, in some cases, fidelity is higher at larger k values, where more matches occur in the later parts of the rankings rather than at the beginning.

Regarding the **edit distance** between the counterfactual and the source sequence (reported in the Distance@k field across all tables), it is evident that most of GENE's solutions differ by only a single element from the original sequence, reaching a maximum of 2 in the most challenging targeted-uncategorized setting.

A different trend emerges for the PACE approach, where the average distance is often below 1, indicating that in many cases, the method outputs the same sequence as the counterfactual. This can be attributed to the automaton's limited ability to accurately approximate the neighborhood of the sequential recommender model, as further evidenced by the lower overall fidelity values.

Target	Model	Dataset	Method	Fidelity@k				Distance@k			
				@1	@5	@10	@20	@1	@5	@10	@20
Drama	BERT4Rec	ml-100k	GENE	1.0	0.648	0.534	0.42	1.002	1.0	1.0	1.0
Drama	BERT4Rec	ml-100k	PACE	0.585	0.469	0.405	0.327	0.4	0.412	0.401	0.442
Drama	GRU4Rec	ml-100k	GENE	1.0	0.603	0.488	0.397	1.006	1.007	1.007	1.0
Drama	GRU4Rec	ml-100k	PACE	0.512	0.426	0.383	0.319	0.402	0.4	0.41	0.402
Drama	SASRec	ml-100k	GENE	1.0	0.715	0.63	0.552	1.0	1.0	1.0	1.0
Drama	SASRec	ml-100k	PACE	0.595	0.512	0.462	0.435	0.345	0.356	0.335	0.333
Action	BERT4Rec	ml-100k	GENE	1.0	0.758	0.632	0.523	1.0	1.0	1.0	1.0
Action	BERT4Rec	ml-100k	PACE	0.476	0.467	0.416	0.372	0.41	0.443	0.413	0.407
Action	GRU4Rec	ml-100k	GENE	1.0	0.789	0.657	0.547	1.007	1.003	1.006	1.002
Action	GRU4Rec	ml-100k	PACE	0.375	0.411	0.401	0.365	0.404	0.397	0.397	0.384
Action	SASRec	ml-100k	GENE	1.0	0.776	0.644	0.525	1.0	1.0	1.0	1.0
Action	SASRec	ml-100k	PACE	0.403	0.439	0.39	0.359	0.413	0.386	0.378	0.383
Adventure	BERT4Rec	ml-100k	GENE	0.997	0.594	0.353	0.209	1.023	1.016	1.012	1.02
Adventure	BERT4Rec	ml-100k	PACE	0.294	0.233	0.159	0.097	0.491	0.518	0.487	0.484
Adventure	GRU4Rec	ml-100k	GENE	1.0	0.628	0.35	0.166	1.033	1.015	1.003	1.0
Adventure	GRU4Rec	ml-100k	PACE	0.238	0.225	0.139	0.084	0.536	0.557	0.565	0.57
Adventure	SASRec	ml-100k	GENE	0.999	0.626	0.403	0.216	1.008	1.005	1.0	1.0
Adventure	SASRec	ml-100k	PACE	0.268	0.22	0.163	0.098	0.573	0.551	0.526	0.457
Horror	BERT4Rec	ml-100k	GENE	0.993	0.358	0.191	0.139	1.077	1.08	1.072	1.046
Horror	BERT4Rec	ml-100k	PACE	0.176	0.081	0.052	0.036	0.735	0.711	0.633	0.529
Horror	GRU4Rec	ml-100k	GENE	0.994	0.163	0.029	0.012	1.2	1.11	1.0	1.0
Horror	GRU4Rec	ml-100k	PACE	0.103	0.028	0.012	0.012	0.608	0.615	0.545	0.455
Horror	SASRec	ml-100k	GENE	0.995	0.232	0.094	0.054	1.097	1.041	1.022	1.0
Horror	SASRec	ml-100k	PACE	0.151	0.046	0.032	0.023	0.683	0.395	0.267	0.182
Animation	BERT4Rec	ml-100k	GENE	0.943	0.432	0.245	0.158	1.143	1.133	1.108	1.107
Animation	BERT4Rec	ml-100k	PACE	0.146	0.088	0.063	0.037	0.87	0.892	0.898	0.943
Animation	GRU4Rec	ml-100k	GENE	0.981	0.238	0.07	0.016	1.319	1.263	1.167	1.267
Animation	GRU4Rec	ml-100k	PACE	0.099	0.036	0.012	0.005	0.806	0.706	0.545	0.4
Animation	SASRec	ml-100k	GENE	0.994	0.357	0.188	0.115	1.113	1.086	1.102	1.102
Animation	SASRec	ml-100k	PACE	0.154	0.069	0.046	0.027	0.855	0.923	0.977	0.96
Fantasy	BERT4Rec	ml-100k	GENE	0.893	0.256	0.018	0.0	1.401	1.382	1.235	nan
Fantasy	BERT4Rec	ml-100k	PACE	0.128	0.037	0.0	0.0	0.851	0.914	nan	nan
Fantasy	GRU4Rec	ml-100k	GENE	0.913	0.317	0.018	0.001	1.628	1.612	1.706	1.0
Fantasy	GRU4Rec	ml-100k	PACE	0.041	0.019	0.002	0.0	0.769	0.778	1.0	nan
Fantasy	SASRec	ml-100k	GENE	0.931	0.354	0.029	0.0	1.34	1.34	1.04	nan
Fantasy	SASRec	ml-100k	PACE	0.141	0.05	0.002	0.0	0.894	0.864	1.0	nan
Average	**BERT4Rec**	**ml-100k**	**GENE**	**0.971**	**0.508**	**0.329**	**0.242**	**1.108**	**1.102**	**1.071**	**1.035**
Average	**BERT4Rec**	**ml-100k**	**PACE**	**0.301**	**0.229**	**0.182**	**0.145**	**0.626**	**0.648**	**0.566**	**0.561**
Average	**GRU4Rec**	**ml-100k**	**GENE**	**0.981**	**0.456**	**0.269**	**0.19**	**1.199**	**1.168**	**1.148**	**1.045**
Average	**GRU4Rec**	**ml-100k**	**PACE**	**0.228**	**0.191**	**0.158**	**0.131**	**0.588**	**0.576**	**0.577**	**0.442**
Average	**SASRec**	**ml-100k**	**GENE**	**0.986**	**0.51**	**0.331**	**0.244**	**1.093**	**1.079**	**1.027**	**1.02**
Average	**SASRec**	**ml-100k**	**PACE**	**0.285**	**0.223**	**0.182**	**0.157**	**0.627**	**0.579**	**0.58**	**0.463**
Results for ml-1m Dataset											
Drama	BERT4Rec	ml-1m	GENE	1.0	0.64	0.49	0.42	1.0	1.0	1.0	1.0
Drama	BERT4Rec	ml-1m	PACE	0.54	0.41	0.365	0.3	0.444	0.512	0.452	0.383
Action	BERT4Rec	ml-1m	GENE	1.0	0.82	0.665	0.565	1.005	1.0	1.0	1.0
Action	BERT4Rec	ml-1m	PACE	0.45	0.43	0.385	0.38	0.256	0.291	0.26	0.276
Adventure	BERT4Rec	ml-1m	GENE	1.0	0.66	0.44	0.305	1.02	1.023	1.011	1.0
Adventure	BERT4Rec	ml-1m	PACE	0.34	0.34	0.23	0.17	0.544	0.559	0.5	0.382
Animation	BERT4Rec	ml-1m	GENE	0.88	0.455	0.335	0.285	1.216	1.154	1.09	1.088
Animation	BERT4Rec	ml-1m	PACE	0.26	0.18	0.15	0.12	0.865	0.861	0.833	0.833
Fantasy	BERT4Rec	ml-1m	GENE	0.925	0.44	0.2	0.145	1.146	1.148	1.2	1.31
Fantasy	BERT4Rec	ml-1m	PACE	0.185	0.1	0.065	0.045	0.946	0.9	0.846	0.778
Average	**BERT4Rec**	**ml-1m**	**GENE**	**0.961**	**0.603**	**0.426**	**0.344**	**1.077**	**1.065**	**1.06**	**1.08**
Average	**BERT4Rec**	**ml-1m**	**PACE**	**0.355**	**0.292**	**0.239**	**0.203**	**0.611**	**0.625**	**0.578**	**0.53**

Table 6.2. Average model fidelity@k and cost@k for k=1,5,10,20 in the **Targeted-Categorized** setting

Target	Model	Dataset	Method	Fidelity@k				Distance@k			
				@1	@5	@10	@20	@1	@5	@10	@20
50	BERT4Rec	ml-100k	GENE	0.589	0.589	0.589	0.589	1.463	1.463	1.463	1.463
50	BERT4Rec	ml-100k	PACE	0.111	0.141	0.141	0.141	0.905	0.925	0.925	0.925
50	GRU4Rec	ml-100k	GENE	0.503	0.503	0.503	0.503	1.776	1.776	1.776	1.776
50	GRU4Rec	ml-100k	PACE	0.043	0.064	0.064	0.064	0.951	0.95	0.95	0.95
50	SASRec	ml-100k	GENE	0.547	0.547	0.547	0.547	1.446	1.446	1.446	1.446
50	SASRec	ml-100k	PACE	0.117	0.142	0.142	0.142	0.945	0.94	0.94	0.94
411	BERT4Rec	ml-100k	GENE	0.071	0.071	0.071	0.071	2.463	2.463	2.463	2.463
411	BERT4Rec	ml-100k	PACE	0.013	0.017	0.017	0.017	1.0	1.0	1.0	1.0
411	GRU4Rec	ml-100k	GENE	0.161	0.161	0.161	0.161	2.414	2.414	2.414	2.414
411	GRU4Rec	ml-100k	PACE	0.023	0.029	0.029	0.029	1.0	1.0	1.0	1.0
411	SASRec	ml-100k	GENE	0.35	0.35	0.35	0.35	1.876	1.876	1.876	1.876
411	SASRec	ml-100k	PACE	0.181	0.204	0.204	0.204	1.0	1.0	1.0	1.0
630	BERT4Rec	ml-100k	GENE	0.284	0.284	0.284	0.284	1.899	1.899	1.899	1.899
630	BERT4Rec	ml-100k	PACE	0.034	0.049	0.049	0.049	1.0	1.0	1.0	1.0
630	GRU4Rec	ml-100k	GENE	0.147	0.147	0.147	0.147	2.453	2.453	2.453	2.453
630	GRU4Rec	ml-100k	PACE	0.004	0.007	0.007	0.007	0.75	0.857	0.857	0.857
630	SASRec	ml-100k	GENE	0.45	0.45	0.45	0.45	1.87	1.87	1.87	1.87
630	SASRec	ml-100k	PACE	0.111	0.142	0.142	0.142	1.0	1.0	1.0	1.0
1305	BERT4Rec	ml-100k	GENE	0.018	0.018	0.018	0.018	2.0	2.0	2.0	2.0
1305	BERT4Rec	ml-100k	PACE	0.001	0.001	0.001	0.001	1.0	1.0	1.0	1.0
1305	GRU4Rec	ml-100k	GENE	0.015	0.015	0.015	0.015	2.0	2.0	2.0	2.0
1305	GRU4Rec	ml-100k	PACE	0.0	0.0	0.0	0.0	nan	nan	nan	nan
1305	SASRec	ml-100k	GENE	0.023	0.023	0.023	0.023	2.227	2.227	2.227	2.227
1305	SASRec	ml-100k	PACE	0.011	0.011	0.011	0.011	1.0	1.0	1.0	1.0
Average	**BERT4Rec**	**ml-100k**	**GENE**	**0.24**	**0.24**	**0.24**	**0.24**	**1.956**	**1.956**	**1.956**	**1.956**
Average	**BERT4Rec**	**ml-100k**	**PACE**	**0.04**	**0.052**	**0.052**	**0.052**	**0.976**	**0.981**	**0.981**	**0.981**
Average	**GRU4Rec**	**ml-100k**	**GENE**	**0.206**	**0.206**	**0.206**	**0.206**	**2.161**	**2.161**	**2.161**	**2.161**
Average	**GRU4Rec**	**ml-100k**	**PACE**	**0.017**	**0.025**	**0.025**	**0.025**	**0.9**	**0.936**	**0.936**	**0.936**
Average	**SASRec**	**ml-100k**	**GENE**	**0.342**	**0.342**	**0.342**	**0.342**	**1.855**	**1.855**	**1.855**	**1.855**
Average	**SASRec**	**ml-100k**	**PACE**	**0.105**	**0.125**	**0.125**	**0.125**	**0.986**	**0.985**	**0.985**	**0.985**
Results for ml-1m Dataset											
2858	BERT4Rec	ml-1m	GENE	0.0	0.0	0.0	0.0	nan	nan	nan	nan
2858	BERT4Rec	ml-1m	PACE	0.0	0.0	0.0	0.0	nan	nan	nan	nan
2005	BERT4Rec	ml-1m	GENE	0.0	0.0	0.0	0.0	nan	nan	nan	nan
2005	BERT4Rec	ml-1m	PACE	0.0	0.0	0.0	0.0	nan	nan	nan	nan
728	BERT4Rec	ml-1m	GENE	0.015	0.015	0.015	0.015	1.0	1.0	1.0	1.0
728	BERT4Rec	ml-1m	PACE	0.01	0.01	0.01	0.01	0.5	0.5	0.5	0.5
2738	BERT4Rec	ml-1m	GENE	0.0	0.0	0.0	0.0	nan	nan	nan	nan
2738	BERT4Rec	ml-1m	PACE	0.0	0.0	0.0	0.0	nan	nan	nan	nan
Average	**BERT4Rec**	**ml-1m**	**GENE**	**0.003**	**0.003**	**0.003**	**0.003**	**1.0**	**1.0**	**1.0**	**1.0**
Average	**BERT4Rec**	**ml-1m**	**PACE**	**0.002**	**0.002**	**0.002**	**0.002**	**0.5**	**0.5**	**0.5**	**0.5**

Table 6.3. Average model fidelity@k and cost@k for k=1,5,10,20 in the **Targeted-Uncategorized** setting

Model	Dataset	Method	Fidelity@k				Distance@k			
			@1	@5	@10	@20	@1	@5	@10	@20
BERT4Rec	ml-100k	GENE	1.0	0.785	0.756	0.739	1.001	1.0	1.0	1.0
BERT4Rec	ml-100k	PACE	0.683	0.689	0.703	0.723	1.0	1.0	1.0	1.0
GRU4Rec	ml-100k	GENE	1.0	0.721	0.663	0.628	1.003	1.0	1.0	1.0
GRU4Rec	ml-100k	PACE	0.675	0.728	0.748	0.762	1.0	1.0	1.0	1.0
SASRec	ml-100k	GENE	1.0	0.768	0.666	0.628	1.0	1.0	1.0	1.0
SASRec	ml-100k	PACE	0.892	0.907	0.911	0.913	1.0	1.0	1.0	1.0
Results for ml-1m Dataset										
BERT4Rec	ml-1m	GENE	1.0	0.695	0.61	0.58	1.01	1.014	1.016	1.017
BERT4Rec	ml-1m	PACE	0.79	0.895	0.95	0.975	1.0	1.0	1.0	1.0

Table 6.4. Average model fidelity@k and cost@k for k=1,5,10,20 in the **Untargeted-Categorized** setting

Model	Dataset	Method	Fidelity@k				Distance@k			
			@1	@5	@10	@20	@1	@5	@10	@20
BERT4Rec	ml-100k	GENE	1.0	0.803	0.826	0.875	1.0	1.0	1.0	1.0
BERT4Rec	ml-100k	PACE	0.741	0.797	0.837	0.862	1.0	1.0	1.0	1.0
GRU4Rec	ml-100k	GENE	1.0	0.871	0.9	0.952	1.0	1.0	1.0	1.0
GRU4Rec	ml-100k	PACE	0.548	0.584	0.614	0.617	1.0	1.0	1.0	1.0
SASRec	ml-100k	GENE	1.0	0.745	0.758	0.81	1.0	1.0	1.0	1.0
SASRec	ml-100k	PACE	0.69	0.704	0.707	0.707	1.0	1.0	1.0	1.0
Results for ml-1m Dataset										
BERT4Rec	ml-1m	GENE	1.0	0.755	0.79	0.855	1.0	1.0	1.0	1.0
BERT4Rec	ml-1m	PACE	0.785	0.855	0.885	0.92	1.0	1.0	1.0	1.0

Table 6.5. Average model fidelity@k and cost@k for k=1,5,10,20 in the **Untargeted-Uncategorized** setting

6.3 Automata evaluation

Evaluating the reliability of the automaton in representing the counterfactual states of sequences in the source sequence's neighborhood is critical for selecting optimal hyperparameters. The purpose of this evaluation is to identify the best hyperparameters for automata learning, and to quantify the automaton's interpretability of counterfactual states in the source sequence's neighborhood.

The evaluation metrics used are **precision**, **accuracy**, and **recall**, computed through the following steps:

1. **Generate a training dataset** of "good" and "bad" points using a genetic algorithm.
2. **Train the automaton** using the RPNI automata learning algorithm.
3. **Generate a test dataset**, ensuring mutations only replace characters **within the alphabet** of the learned automaton.
4. **Evaluate the automaton** using the **test dataset**:
    - For each *good* sequence, determine if the automaton classifies it correctly as a *true positive* or misclassifies it as a *false negative*.
    - For each *bad* sequence, determine if the automaton classifies it correctly as a *true negative* or misclassifies it as a *false positive*.
5. **Repeat the evaluation** across multiple source sequences (users).
6. **Compute final metrics** by aggregating all true positives, false positives, true negatives, and false negatives across all evaluated sequences within a given hyperparameter set.

6.3.1 Evaluation Results

Using the same hyperparameter configuration from the counterfactual generation hyperparameter tuning, we conducted an evaluation on **400 users** of the **MovieLens 100K** dataset, using the **BERT4Rec** model, and maintaining the same **representative targets** in the targeted setting. The summary of the evaluation results is presented in the following tables.

> **Note:** We reused the previously tuned hyperparameters because the counterfactual generation process inherently includes the automata learning process. The underlying assumption is that *"good counterfactual generation results imply good automata learning metrics"* and vice versa. Ideally, conducting a separate hyperparameter tuning for automata learning would have been optimal. However, due to resource constraints, this was not feasible.

The evaluation results highlight an **interesting trend** in automata learning performance across different categories. Less frequent categories, such as Fantasy, exhibit higher precision, accuracy, and recall compared to more common categories like Drama. At first glance, this might seem counterintuitive, since larger datasets typically lead to more robust

Target	Precision		Accuracy		Recall	
	Aggregated	Per-User	Aggregated	Per-User	Aggregated	Per-User
Drama	0.728	0.7332	0.5292	0.5290	0.1548	0.1530
Action	0.723	0.7402	0.5159	0.5154	0.1663	0.1640
Adventure	0.7659	0.7752	0.5241	0.5238	0.2093	0.2063
Horror	0.7746	0.7861	0.5704	0.5767	0.3998	0.3973
Animation	0.8159	0.7873	0.5804	0.5767	0.4332	0.4184
Fantasy	0.8769	0.7777	0.6278	0.5850	0.5666	0.5062
Average	**0.7950**	**0.7609**	**0.5507**	**0.5476**	**0.3086**	**0.3081**

Table 6.6. Automata learning evaluation for the targeted-categorized setting. The *Aggregated* value in each cell represents the evaluation computed by aggregating TP, FP, TN, and FN across all users before deriving the evaluation metrics. The *Per-User* corresponds to the alternative evaluation method, where metrics were first computed per user and then averaged to account for empty datasets.

Target	Precision		Accuracy		Recall	
	Aggregated	Per-User	Aggregated	Per-User	Aggregated	Per-User
50	0.8860	0.4480	0.6202	0.3317	0.5268	0.2701
411	0.9766	0.0402	0.8616	0.0366	0.8695	0.0364
630	0.9384	0.2348	0.7268	0.1905	0.7227	0.1822
1305	0.9680	0.0071	0.7789	0.0059	0.7954	0.0060
Average	**0.9111**	**0.1825**	**0.6595**	**0.1491**	**0.6045**	**0.1275**

Table 6.7. Automata learning evaluation for the targeted-uncategorized setting.

Setting	Precision		Accuracy		Recall	
	Aggregated	Per-User	Aggregated	Per-User	Aggregated	Per-User
Uncategorized	0.8149	0.7575	0.5149	0.5137	0.2158	0.1962
Categorized	0.7895	0.7892	0.5987	0.5976	0.2887	0.2855

Table 6.8. Automata learning evaluation for the untargeted settings.

learning. However, the key factor influencing these results lies in the dataset generation

process. The genetic algorithm used to construct the dataset sometimes fails to produce usable data for rare categories due to the limited number of sequences available. Because this algorithm relies on generating both positive and negative examples, when a target category is infrequent, the dataset may end up **empty**, preventing the automaton from being learned at all.

This limitation introduces a bias in the evaluation metrics. In the initial computation method—represented by the *Aggregated* column in the evaluation tables— precision, accuracy, and recall were derived by aggregating the number of true positives, false positives, true negatives, and false negatives across all users before calculating the final scores. This approach considers only those instances where a dataset was successfully generated, since the aggregated values are zero if no dataset is available. As seen in Table 6.6, Fantasy, despite being a less frequent category, achieved notably higher precision, accuracy, and recall compared to Drama, a result that does not necessarily indicate superior automata learning but rather an artifact of the dataset generation process.

To mitigate this issue, an alternative evaluation method was introduced—represented by the *Per-User* column in the evaluation tables. Instead of aggregating across all users before computing precision, accuracy, and recall, these metrics were first calculated per user and then averaged. This modification ensured that users with an empty dataset contributed to the final results, preventing artificially inflated scores for rare categories. The values obtained with this alternative approach confirm the previous bias. Categories that previously displayed high performance, such as Fantasy, now show significantly lower scores. This confirms that the original method masked the difficulty of learning from sparse datasets by excluding failed cases from the metric computation. It has to be noted that less popular categories may still generate a highly imbalanced test dataset, hence the metrics for those may still contain some bias.

Examining the average values across different settings provides insight into the overall performance of automata learning. The targeted-categorized setting (Table 6.6), along with both categorized and uncategorized cases in the untargeted setting (Table 6.8), demonstrates consistently high precision across both evaluation methods, moderate accuracy, and notably low recall. This indicates a tendency for the automata to misclassify numerous sequences as counterfactuals, even when they are not. In contrast, the targeted-uncategorized setting (Table 6.7) initially appears to perform well under the first evaluation method, which does not account for empty datasets. However, since targeting a single item results in an extremely sparse solution space, the majority of generated datasets remain empty. This is reflected in the second evaluation method, where performance scores drop significantly once empty datasets are properly considered, revealing the inherent limitations of this approach.

Chapter 7

Conclusions

7.1 GENE vs. PACE

While the **PACE** method is highly powerful, its accuracy in finding a valid and good counterfactual primarily depends on how well the automaton represents the counterfactual state of the sequence after applying a particular edit.

It is intuitive that counterfactual generation using **GENE** alone will often outperform PACE:

- If GENE fails to find any counterfactual, PACE will also fail, since the training data for the automata is generated by GENE.
- If GENE finds only a few (or even just one) valid counterfactual, the result for that instance will be a success, while PACE may still fail if the dataset doesn't adequately represent the recommender behavior in the neighborhood of the source sequence.
- If GENE finds many valid counterfactuals, the likelihood of having discovered an optimal counterfactual is higher than PACE's chances of doing so.

One significant advantage of using GENE alone is that it allows for the generation of only the **bad** dataset (composed of counterfactuals). The **good** dataset, which serves as counterexamples for the RPNI algorithm in the automaton learning process, is not required for counterfactual generation. Furthermore, with GENE, we can generate fewer points, as each point in the bad dataset will necessarily be a counterfactual. While a larger number of points —8192 for each part of the dataset—is needed to train a good automaton with RPNI, with GENE, a higher number of points simply increases the probability of finding a counterfactual closer to the original sequence.

On the other hand, PACE offers **more transparency** in the counterfactual generation process. It provides an insightful view of how each possible edit (within a certain distance) to the original sequence results in either a factual or counterfactual example, just by examining the automaton states and transitions, which represent the sequence counterfactual state

and how does it change according to the possible edits. The process is more interpretable, enabling users to understand why certain edits lead to a valid counterfactual.

In the final application, the user should have the option to decide whether to use GENE alone or the full PACE approach. GENE should be employed for quick counterfactual generation, especially in cases where speed is essential. The full approach, incorporating PACE, should be used when a more thorough and insightful explanation is needed, due to the higher number of counterfactuals generated by GENE and the comprehensive overview provided by PACE.

7.2 Practical Implications and Use Cases

The findings of this thesis have direct applications in various domains where sequential recommendation systems play a crucial role. Integrating explainable AI into systems such as e-commerce, content streaming, healthcare, financial and educational platforms fosters greater user trust and decision-making confidence. Additionally, counterfactual explanations assist developers in debugging recommendation models by identifying the key interactions influencing predictions, while businesses can leverage these insights to refine their recommendation strategies and align them with user expectations.

7.3 Limitations and Future Developments

Despite the advancements in counterfactual explanations, certain limitations persist:

1. The computational cost of PACE remains a challenge, necessitating optimizations for better performance.
2. There is a trade-off between interpretability and accuracy, as PACE provides greater transparency while GENE ensures higher precision.
3. Scalability is another concern, as real-time counterfactual generation in large-scale recommender systems may be impractical.
4. Further improvements in hyperparameter tuning may significantly enhance PACE's performance by refining the automata learning process and optimizing the path-search mechanism, leading to more accurate and efficient counterfactual generation.
5. Additionally, GENE is easily extendable for generating multiple counterfactuals, though their diversity is not explicitly ensured, whereas PACE currently produces only a single counterfactual per run. Addressing this limitation in PACE by enabling the generation of multiple, diverse counterfactuals could enhance its applicability.

7.4 Conclusions

This thesis introduced a novel approach to counterfactual explanations in sequential recommendation systems through genetic algorithms (**GENE**) and automata learning-based methods (**PACE**). The experiments demonstrated that GENE provides **high-fidelity**

counterfactual generation, allowing it to explain most data points, whereas PACE enhances **interpretability** by modeling sequential data through **automata learning**. The choice between the two depends on the trade-off between **computational efficiency**, **accuracy** and **transparency**. The framework was thoroughly evaluated across four settings: **targeted** and **untargeted**, **categorized** and **uncategorized**, ensuring a comprehensive analysis of its performance. The proposed methods have applications in various industries, including e-commerce, content streaming, finance, healthcare, and education, where explainability strengthens user trust and engagement. While computational complexity and scalability present challenges, this research establishes a foundation for further improvements in interpretable recommendation systems. Optimizing these methods and improving user accessibility will contribute to more transparent AI-driven decision-making systems.

Bibliography

A. Adriansyah, B.F. Van Dongen, and W.M.P. Van Der Aalst. Conformance Checking Using Cost-Based Fitness Analysis. In *2011 IEEE 15th International Enterprise Distributed Object Computing Conference*, pages 55–64, Helsinki, Finland, August 2011. IEEE. ISBN 978-1-4577-0362-1. doi: 10.1109/EDOC.2011.12.

Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, November 1987. ISSN 08905401. doi: 10.1016/0890-5401(87)90052-6.

Filippo Betello, Federico Siciliano, Pushkar Mishra, and Fabrizio Silvestri. Investigating the Robustness of Sequential Recommender Systems Against Training Data Perturbations, December 2023.

Jesús Bobadilla, Fernando Ortega, Antonio Hernando, and Jesús Bernal. A collaborative filtering approach to mitigate the new user cold start problem. *Knowledge-Based Systems*, 26:225–238, February 2012. ISSN 0950-7051. doi: 10.1016/j.knosys.2011.07.021.

John S. Breese, David Heckerman, and Carl Kadie. Empirical Analysis of Predictive Algorithms for Collaborative Filtering, January 2013.

Giuseppe De Giacomo, Fabrizio Maria Maggi, Andrea Marrella, and Fabio Patrizi. On the Disruptive Effectiveness of Automated Planning for LTLf-Based Trace Alignment. *Proceedings of the AAAI Conference on Artificial Intelligence*, 31(1), February 2017. ISSN 2374-3468, 2159-5399. doi: 10.1609/aaai.v31i1.11020.

Riccardo Guidotti. Counterfactual explanations and how to find them: Literature review and benchmarking. *Data Mining and Knowledge Discovery*, 38(5):2770–2824, September 2024. ISSN 1384-5810, 1573-756X. doi: 10.1007/s10618-022-00831-6.

Riccardo Guidotti, Anna Monreale, Salvatore Ruggieri, Dino Pedreschi, Franco Turini, and Fosca Giannotti. Local Rule-Based Explanations of Black Box Decision Systems, May 2018.

F. Maxwell Harper and Joseph A. Konstan. The MovieLens Datasets: History and Context. *ACM Transactions on Interactive Intelligent Systems*, 5(4):1–19, January 2016. ISSN 2160-6455, 2160-6463. doi: 10.1145/2827872.

Balázs Hidasi and Alexandros Karatzoglou. Recurrent Neural Networks with Top-k Gains for Session-based Recommendations. In *Proceedings of the 27th ACM International*

Conference on Information and Knowledge Management, Torino Italy, October 2018. ACM. doi: 10.1145/3269206.3271761.

Balázs Hidasi, Alexandros Karatzoglou, Linas Baltrunas, and Domonkos Tikk. Session-based Recommendations with Recurrent Neural Networks, March 2016.

Wang-Cheng Kang and Julian McAuley. Self-Attentive Sequential Recommendation, August 2018.

Amir-Hossein Karimi, Gilles Barthe, Borja Balle, and Isabel Valera. Model-Agnostic Counterfactual Explanations for Consequential Decisions.

Luca Longo, Mario Brcic, Federico Cabitza, Jaesik Choi, Roberto Confalonieri, Javier Del Ser, Riccardo Guidotti, Yoichi Hayashi, Francisco Herrera, Andreas Holzinger, Richard Jiang, Hassan Khosravi, Freddy Lecue, Gianclaudio Malgieri, Andrés Páez, Wojciech Samek, Johannes Schneider, Timo Speith, and Simone Stumpf. Explainable Artificial Intelligence (XAI) 2.0: A manifesto of open challenges and interdisciplinary research directions. *Information Fusion*, 106:102301, June 2024. ISSN 15662535. doi: 10.1016/j.inffus.2024.102301.

Scott M Lundberg and Su-In Lee. A Unified Approach to Interpreting Model Predictions.

Ramaravind K. Mothilal, Amit Sharma, and Chenhao Tan. Explaining machine learning classifiers through diverse counterfactual explanations. In *Proceedings of the 2020 Conference on Fairness, Accountability, and Transparency*, pages 607–617, Barcelona Spain, January 2020. ACM. ISBN 978-1-4503-6936-7. doi: 10.1145/3351095.3372850.

José Oncina and Pedro García. *IDENTIFYING REGULAR LANGUAGES IN POLYNOMIAL TIME*, volume 5, pages 99–108. WORLD SCIENTIFIC, February 1993. ISBN 978-981-279-791-9. doi: 10.1142/9789812797919_0007.

Rafael Poyiadzi, Kacper Sokol, Raul Santos-Rodriguez, Tijl De Bie, and Peter Flach. FACE: Feasible and Actionable Counterfactual Explanations. In *Proceedings of the AAAI/ACM Conference on AI, Ethics, and Society*, pages 344–350, New York NY USA, February 2020. ACM. ISBN 978-1-4503-7110-0. doi: 10.1145/3375627.3375850.

Massimo Quadrana, Paolo Cremonesi, and Dietmar Jannach. Sequence-Aware Recommender Systems, February 2018.

Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. "Why Should I Trust You?": Explaining the Predictions of Any Classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1135–1144, San Francisco California USA, August 2016. ACM. ISBN 978-1-4503-4232-2. doi: 10.1145/2939672.2939778.

Francesco Ricci, Lior Rokach, Bracha Shapira, and Paul B. Kantor, editors. *Recommender Systems Handbook*. Springer US, Boston, MA, 2011. ISBN 978-0-387-85819-7 978-0-387-85820-3. doi: 10.1007/978-0-387-85820-3.

Deepjyoti Roy and Mala Dutta. A systematic review and research perspective on rec-

ommender systems. *Journal of Big Data*, 9(1):59, May 2022. ISSN 2196-1115. doi: 10.1186/s40537-022-00592-5.

Ruslan Salakhutdinov, Andriy Mnih, and Geoffrey Hinton. Restricted Boltzmann machines for collaborative filtering. In *Proceedings of the 24th International Conference on Machine Learning*, pages 791–798, Corvalis Oregon USA, June 2007. ACM. ISBN 978-1-59593-793-3. doi: 10.1145/1273496.1273596.

Maximilian Schleich, Zixuan Geng, Yihong Zhang, and Dan Suciu. GeCo: Quality Counterfactual Explanations in Real Time, May 2021.

Suvash Sedhain, Aditya Krishna Menon, Scott Sanner, and Lexing Xie. AutoRec: Autoencoders Meet Collaborative Filtering. In *Proceedings of the 24th International Conference on World Wide Web*, pages 111–112, Florence Italy, May 2015. ACM. ISBN 978-1-4503-3473-0. doi: 10.1145/2740908.2742726.

Shubham Sharma, Jette Henderson, and Joydeep Ghosh. CERTIFAI: Counterfactual Explanations for Robustness, Transparency, Interpretability, and Fairness of Artificial Intelligence models. In *Proceedings of the AAAI/ACM Conference on AI, Ethics, and Society*, pages 166–172, February 2020. doi: 10.1145/3375627.3375812.

Fei Sun, Jun Liu, Jian Wu, Changhua Pei, Xiao Lin, Wenwu Ou, and Peng Jiang. BERT4Rec: Sequential Recommendation with Bidirectional Encoder Representations from Transformer, August 2019.

Khanh Hiep Tran, Azin Ghazimatin, and Rishiraj Saha Roy. Counterfactual Explanations for Neural Recommenders. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 1627–1631, July 2021. doi: 10.1145/3404835.3463005.

Stratis Tsirtsis and Manuel Gomez Rodriguez. Decisions, Counterfactual Explanations and Strategic Behavior. In *Advances in Neural Information Processing Systems*, volume 33, pages 16749–16760. Curran Associates, Inc., 2020.

Sandra Wachter, Brent Mittelstadt, and Chris Russell. Counterfactual Explanations without Opening the Black Box: Automated Decisions and the GDPR, March 2018a.

Sandra Wachter, Brent Mittelstadt, and Chris Russell. Counterfactual Explanations without Opening the Black Box: Automated Decisions and the GDPR, March 2018b.

Yao Wu, Christopher DuBois, Alice X. Zheng, and Martin Ester. Collaborative Denoising Auto-Encoders for Top-N Recommender Systems. In *Proceedings of the Ninth ACM International Conference on Web Search and Data Mining*, pages 153–162, San Francisco California USA, February 2016. ACM. ISBN 978-1-4503-3716-8. doi: 10.1145/2835776.2835837.

Shengyu Zhang, Dong Yao, Zhou Zhao, Tat-Seng Chua, and Fei Wu. CauseRec: Counterfactual User Sequence Synthesis for Sequential Recommendation. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 367–377, Virtual Event Canada, July 2021. ACM. ISBN 978-1-4503-8037-9. doi: 10.1145/3404835.3462908.

Shuai Zhang, Lina Yao, Aixin Sun, and Yi Tay. Deep Learning Based Recommender System: A Survey and New Perspectives. *ACM Computing Surveys*, 52(1):1–38, January 2020. ISSN 0360-0300, 1557-7341. doi: 10.1145/3285029.