MLOps in Recommender Systems: Building a Continuous Training Pipeline with Concept Drift Awareness

Bachelorarbeit

im Studiengang
Wirtschaftsinformatik und digitale Medien

vorgelegt von

Sebastian Sätzler
Matr.-Nr.: 37635

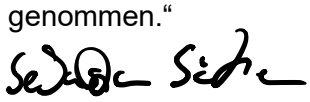
am 18. Juli 2022
an der Hochschule der Medien Stuttgart

Erstprüfer/in:	Prof. Dr. Jan Kirenz
Zweitprüfer/in:	Prof. Dr. Hendrik Meth

Ehrenwörtliche Erklärung

„Hiermit versichere ich, Sebastian Sätzler, ehrenwörtlich, dass ich die vorliegende Bachelorarbeit (bzw. Masterarbeit) mit dem Titel: „MLOps in Recommender Systems: Building a Continuous Training Pipeline with Concept Drift Awareness“ selbstständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden.

Ich habe die Bedeutung der ehrenwörtlichen Versicherung und die prüfungsrechtlichen Folgen (§ 26 Abs. 2 Bachelor-SPO (6 Semester), § 24 Abs. 2 Bachelor-SPO (7 Semester), § 23 Abs. 2 Master-SPO (3 Semester) bzw. § 19 Abs. 2 Master-SPO (4 Semester und berufsbegleitend) der HdM) einer unrichtigen oder unvollständigen ehrenwörtlichen Versicherung zur Kenntnis genommen.“

Aichschieß, 8.8.2022, 

Kurzfassung

Mit dem vermehrten Einsatz von maschinellem Lernen (ML) im Unternehmenssektor, sind Unternehmen besonderen Herausforderungen ausgesetzt, die es zu überwinden gilt. ML-Systeme werden in Produktivumgebungen mit unvorhersehbaren Änderungen in den Daten konfrontiert und müssen sich den Anforderungen des iterativen Softwareentwicklungsprozesses fügen. Um diese Voraussetzungen zu erfüllen, bedarf es einen Paradigmenwechsel, wie ML-Systeme entwickelt und betrieben werden sollen. Das aufkommende Gebiet der MLOps verfolgt das Ziel, Praktiken einzuführen, die das Arbeiten mit ML effizienter und wartbarer gestalten. MLOps ist an die Prinzipien des DevOps angelehnt, welche auf die speziellen Bedürfnisse des maschinellen Lernens angepasst werden. Dieses Forschungsprojekt widmet sich der Untersuchung des aktuellen Standes der Technik im Bereich der MLOps. So soll unter Verwendung von TensorFlow Extended und Apache Airflow und Berücksichtigung der Design-Science-Research Methodik eine MLOps Pipeline entworfen und implementiert werden. Das Ergebnis dieser Arbeit ist eine Machbarkeitsstudie einer kontinuierlichen Lernpipeline unter Einsatz eines modernen Empfehlungsdienstes. Diese Pipeline ist konzipiert, um Konzeptdrift durch Degradierung der Modellleistung automatisch zu erkennen und ein erneutes Modelltraining anzustoßen.

Schlagwörter: Pipeline, MLOps, Kontinuierliches Lernen, Konzeptdrift, Empfehlungsdienst, Maschinelles Lernen

Abstract

With the increase in adoption of machine learning (ML) in the enterprise sector, comes a unique set of challenges that companies must face. Machine learning systems in productive environments are subjected to unpredictable changes in the data and need to conform to the iterative cycles of agile software development. In order to meet these requirements, a paradigm shift in the development and operation of ML systems is needed. The emergent field of MLOps sets out to introduce practices to make ML operations more efficient and maintainable. MLOps takes inspiration from principles of DevOps and adapts them to the specific needs of ML. This research project serves as a study of the current state of the art (SotA) for MLOps. Using the Design Science Research methodology, an MLOps pipeline is designed and implemented with TensorFlow Extended and Apache Airflow. The result of this thesis is a proof of concept of a continuous training pipeline for a SotA recommendation system. This pipeline is designed to automatically relearn the recommender system upon detecting concept drift through degradation of model performance.

Keywords: Pipeline, MLOps, Continuous Training, Concept Drift, Recommender System, Machine Learning

Table of Contents

Table of Figures

List of Tables

List of Abbreviations

ML	Machine Learning
GPU	Graphics Processing Unit
CPU	Central Processing Unit
RS	Recommender System
CD	Concept Drift
TFX	TensorFlow Extended
TFRS	TensorFlow Recommenders
DCN	Deep & Cross Network
DD	Data Drift
STEM	Academic disciplines of Science, Technology, Engineering, Mathematics
PoC	Proof of Concept
IS	Information System
SotA	State of the Art
AI	Artificial Intelligence
API	Application Programming Interface
CF	Collaborative Filtering
CB	Content-Based Recommender System
TF-IDF	Term Frequency-Inverse Document Frequency
MF	Matrix Factorization
DL	Deep Learning
DNN	Deep Neural Networks
NN	Neural Network
ANN	Artificial Neural Network
SVD	Singular Value Decomposition
FM	Factorization Model
NLP	Natural Language Processing
ReLU	Rectified Linear Unit
QPS	Queries per Second
CI/CD	Continuous Integration Continuous Delivery

EDA	Exploratory Data Analysis
DAG	Directed Acyclic Graph
PTA	Post-Training Action
RMSE	Root Mean Squared Error

1 Introduction

1.1 Motivation

Over the last two decades Machine Learning (ML) has become one of the fastest growing technical fields. Gartner estimates the total revenue in the ML market to be over 51 billion US-dollars for the year 2021, with an expected growth of 21% in 2022 (Rimol, 2021). It managed to secure a position as one of the top fields in computer science as well as enterprise adoption. ML combines concepts of linear algebra and statistics and applies them to large datasets to find patterns and generalizations in the data, which can be used to make predictions or classifications. Leveraging these complex algorithms with the computational power of modern GPUs and CPUs, ML has been applied in a large variety of sectors ranging from medicine for diagnostics, to transportation for self-driving cars and e-commerce for shopping cart optimization (Choy et al., 2018). The latter sector employs so called Recommender Systems (RS) with the goal of suggesting products that coincide with the taste of the customer. With the advent of e-commerce, RSs have gained increasing interest from academia and especially the enterprise sector (Singh, Choudhury, Dey, & Pramanik, 2021). RSs play a major role in large tech corporations when trying to engage, retain and entice their user-base within their platform (Jannach & Zanker, 2022).

Despite the wide use and success of Recommender Systems and Machine Learning in general, it still is a relatively new field with a lot of research opportunities (Jordan & Mitchell, 2015). While recommender systems are considered integral to many online platforms, their precision and accuracy often lack in comparison to other ML fields. This is, among other things, due to the nature of the data that Recommender Systems work with, which is often sparse (Khusro, Ali, & Ullah, 2016). Consequently, Recommender Systems are especially susceptible to bad data quality and therefore benefit from comprehensive data curation and monitoring. Therefore, RSs lend themselves to data-centric approaches when building, deploying and maintaining them, which is one of the subject matters that the field of MLOps sets out to tackle (Miranda, 2021).

MLOps emerged from the paradigm of DevOps and seeks to apply an automated and standardized approach to the lifecycle of ML applications, similar to what DevOps does for conventional Software. MLOps is geared to the specific needs and problems of ML, such that its practices vary from those of DevOps, while still sharing the same goal of rapid and frequent deployment of Software at high quality (Makinen, Skogstrom, Laaksonen, & Mikkonen, 2021). The effect of data quality on the ML model requires that data quality management is an integral part of every MLOps system, since data quality affects all aspects of the machine learning lifecycle (Renggli et al., 2021). Lack of sufficient data quality can have detrimental effects to the ML system’s performance, which can manifest itself in different ways.

The specific manifestation covered in this research is known as concept drift (CD), which describes a change in the output *y* given a constant input *x* over time (Lu et al., 2018). Deteriorating RS performance due to CD can directly impact the online platform it is used on, as outputs of RSs are generally reciprocated back to the user experience. For instance, if a movie streaming platform doesn't recommend appropriate movies to a user anymore because it fails to adapt to the user's change in taste, the user might stop watching movies on that platform and eventually cancel their subscription.

Issues of Concept Drift need to be addressed and mitigated to ensure user-base retention for online services. Furthermore, CD needs to be incorporated into an MLOps system for maintainability, consistency and automation in a unified process.

The result of this work, called an artifact, will be a proof-of-concept (PoC) of a CD aware MLOps pipeline for a state-of-the-art (SotA) RS. CD awareness relates to the ability to adapt the model to CD in the data. Specifically, a continuous training (CT) pipeline is designed, that automatically retrains an RS model upon detecting CD.

This thesis serves as a thorough documentation of the design of the artifact, which is based on a comprehensive study of scientific literature touching the topics of Recommender Systems, MLOps and Concept Drift. The result will then be qualitatively evaluated and discussed.

This research follows the design science research (DSR) methodology of Alan R. Hevner (2004).

1.2 Research Methodology

Design science is a research paradigm that emerged as a differentiation to natural science in STEM. Natural science, also referred to as behavioral science, is associated with fields like mathematics, physics, biology and chemistry. Its research methodology follows the objective of uncovering facts and theories about reality. Juxtaposed to the natural science lies the design science. Instead of uncovering rules and theories about the nature of reality, design science sets out to engineer and create artifacts with tools from scientific literature. Design science is predominantly represented in the engineering and computer science fields, where PoCs and prototypes are the result of a lot of academic research. Both behavioral science and design science have distinguished approaches on how to conduct research.

Design science research contains a set of frameworks and best practices to manage academic work in the design science department. One of the more prominent methodologies is Alan R. Hevner's "three cycles" of DSR (Hevner et al., 2004). Hevner originally designed his framework to involve the research aspect more closely to the development process of Information Systems (IS) in enterprise environments. It consists of 3 cycles which are closely related to each other and serve to build an artifact. The three cycles are what Hevner argues separates design science from other research paradigms (Hevner, 2007).

The artifact is the eventual product of the academic work using DSR. Since its first publication in 2004, DSR has been applied in a wide variety of fields not only in conventional engineering and computer science. This means that the term “artifact” has a broad definition in general describing anything that emerges from design science research. It could range from a theoretical model that was derived from other academic work to a physical prototype or a production-ready software system.

The goal of DSR is to create an innovative artifact, which incorporates both theoretical-scientific, as well as the practical-environmental (e.g. business) aspects into its design. The iteration through the three cycles creates a mutual feedback-loop between the artifact, the science and business environment (Hevner et al., 2004). The result is an artifact, which is attuned to the business needs of an enterprise, while also holding scientific value and enriching the academic field with new insights and findings.

In the following, the three cycles will be elaborated in more detail.

1. **The relevance cycle**: The relevance cycle initiates the DSR process. In this cycle all requirements relevant to the artifact and the research are worked out. First, a problem is defined and opportunities and arguments are laid out supporting research to resolve the problem with an artifact. Since DSR has its roots in the enterprise sector, it is vital to map out and contextualize the environment this research takes place in, as it directly influences the design of the artifact. In order to evaluate the artifact, acceptance criteria need to be defined. This way a conclusion can be made whether the artifact succeeded in its goals to improve the environment or not (Hevner & Chatterjee, 2010). In this work the results of the relevance cycle are laid out in chapter Environment.

2. **The rigor cycle**: Following the relevance cycle, comes the rigor cycle. While the relevance cycle establishes the requirements for the project, the rigor cycle introduces the methods, drawn from scientific literature, used to create the artifact (Hevner & Chatterjee, 2010). This so-called *knowledge base* consists of engineering methods and scientific theories and sets the foundation from which the artifact will be designed and built from. A thorough rigor cycle ensures that the artifact is grounded in state-of-the-art (SotA) literature from the academic field. This establishes the connection to other scientific contributions and thus sets it apart from routine designs and routine design processes (Hevner & Chatterjee, 2010). The rigor cycle gives the artifact the scientific weight it requires to be acknowledged as an academic contribution, consequently it is vital that the design of the artifact draws sufficiently from the knowledge base of the rigor cycle. The knowledge base is covered in Recommender Systems and MLOps in chapter 2.

3. **The design cycle**: “*The internal design cycle is the heart of any design science research project.*” (Hevner & Chatterjee, 2010) It is the culmination of the rele-

vance and the rigor cycle. The information acquired from the two prior cycles will now be deployed to design and implement the artifact. Hevner points out that it is not possible to retain both maximum relevance and rigor simultaneously, thus a balance between both need to be struck (Hevner & Chatterjee, 2010). In the context of the design cycle, the rigor represents the actual construction of the artifact, meaning the implementation of the knowledge base gained from the rigor cycle. In opposition to the rigor stands the relevance. The relevance represents all the requirements and evaluation criteria that were specified in the relevance cycle. The discrepancy between the relevance and the rigor gets resolved by the artifact, which is the bridge between both and thus constitutes the business and scientific contribution. The design cycle is documented in chapter 3 and 4.

DSR is a non-linear process. With progression of the research project, the relevance, rigor, and design cycle can change as new insights are garnered. Through iterative cycles Hevner's DSR methodology accounts for the often unpredictable nature of the artifact creation process. Should either parts of the relevance, rigor or design fall out of line with the current state of the project, it needs to be updated by reiteration (Hevner et al., 2004).

Once the research is conducted and the artifact is created it itself becomes part of the knowledge base, whose insights can now be used for other research projects. These insights are worked out in the Evaluation chapter and then summarized in the Conclusion chapter at the end of the thesis.

2 Current Environment & State of the Art

2.1 Environment

Machine Learning serves great value to businesses. In 2017, Netflix for instance claimed an estimated saving of $1 billion through their use of RSs (Columbus, 2017). Since 2017, ML algorithms became more sophisticated and hardware more powerful to make Artificial Intelligence (AI[1]) operations more efficient, effective and in return more profitable. Gartner calculated the revenue of the AI software market to be over $51 billion in the year 2021 with a prediction to surpass $60 billion by the end of 2022 (Rimol, 2021). Open-source software, ML cloud services and an active community make AI more accessible to a wide variety of businesses. These developments make a growing number of institutions consider optimizing, augmenting, or even reinventing their current operations with ML. McKinsey's "The state of AI in 2021" reports that 56% of their surveyed businesses have adopted ML and AI in at least one of their business functions. An increase of 6% compared to the preceding year (Chui, Hall, Singla, & Sukharevsky, 2021). It's apparent that ML and AI receive increasing interest in the enterprise sector.

According to a survey conducted by Refinitiv, out of 447 international institutions that use ML, only 46% have deployed AI in multiple areas and are core to its business, whereas 44% deployed ML in pockets, while the remaining 10% were still prototyping and investing in its infrastructure (Baker, 2019). This indicates that a majority of enterprises, while considering or using ML for their businesses, struggle to embed it into their existing infrastructure. This observation is also supported by Algorithmia's 2020 report on enterprise machine learning, which uncovers that 55% of companies "*actively developing machine learning lifecycles or […] beginning their machine learning journey*" (Algorithmia, 2020) have yet to deploy a machine learning model. This report highlights that a lot of the main difficulties of ML lie in its operational aspect, such as reproducibility, versioning of models and scaling of the ML system. This leads to "*unreasonably long roads to deployment*" (Algorithmia, 2020) and impedes evolving the ML system to higher levels of maturity. Another Refinitiv study also identified the lack in data quality as the biggest challenge for ML and data science (Refinitiv, 2020).

The aforementioned McKinsey whitepaper made the observation, that companies most successful with AI were employing advanced operation procedures such as MLOps, as well as putting greater effort into mitigating and reacting to "*AI-related-risks*", such as concept drift. This applies to the operation of RSs as well. Operationalizing the ML lifecycle and ensuring high model quality through data quality assurance are essential

[1] Throughout this paper ML and AI will be used interchangeably.

to an RS in a productive environment. A data-centric approach to RSs combined with the application of MLOps practices should prevent or alleviate unsatisfactory RS performance while ensuring maintainability and reproducibility to a mostly automated ML process. Especially the phenomenon of CD needs to be accounted for as it is an inevitable occurrence for a majority of real-world data. A key challenge for RSs is the often unpredictable nature and sudden appearance of CD, which can be detrimental to business operation. Shift in the data can be subliminal as it is not tangible in most cases and therefore can go unnoticed by Data Scientists and ML engineers. Consequently, the effects of CD can first become apparent through degradation of model performance. Scalable and automated ways to account for change in ML data still are in their infancy and therefore require further research and development to find new solutions. In the long run a SotA ML infrastructure will make RS operations more profitable for businesses through labor reduction and performance improvements.

This introduction poses the baseline for the relevance cycle of Hevner's DSR methodology. The collected insights from the whitepapers serve to map out the environment and establish the motivation for this research. The acceptance criteria for the developed artifact will be in part derived from the findings and motivation in this chapter, which will be elaborated in the following chapter.

Having established the environment and laid out the motivation, the following research question can be seen as central for this work: "*What can an MLOps pipeline for a recommender system, that takes concept drift into account, look like?*"

2.2 Recommender Systems

2.2.1 Overview

RSs are used to uncover user preferences on online platforms. The most known examples are shopping basket recommendations on e-commerce sites like Amazon or movie and video recommendations of streaming platforms like Netflix and YouTube. As a generalization, the term *item* is used to denote any object in a set of items that can be recommended to a user, such as a product or a movie. The recipient of an item is referred to as the *user*. The user-item relationship builds the foundation of any RS as RSs work under the assumption that there exists underlying dependencies between users and items.

The main task of an RS is to correctly identify these dependencies and use them to match appropriate items and users (Aggarwal, 2016). In order to reduce information overload on websites with video and product catalogues, RSs are applied to identify items that will interest the user (Alyari & Jafari Navimipour, 2018). Aggarwal (2016) lists four aspects that define a good recommendation:

1. *Relevance*: The most fundamental objective of an RS is to recommend items that are relevant and interesting to the individual user.

2. *Novelty*: The RS should introduce items to the users that they weren't aware of before. It has been shown that RSs can negatively impact the sales diversity of e-commerce, when only popular items are recommended to users (Fleder & Hosanagar, 2007).
3. *Serendipity*: Similar to the point above, RSs should have the ability to surprise users with somewhat unexpected but relevant items. While novelty focuses on the quantitative aspect of recommending items that aren't on the radar of the user, serendipity considers the qualitative-psychological impact of a surprise in the recommendation.
4. *Increasing recommendation diversity*: This last point envisions that RSs should ensure a diverse set of items in a recommendation feed. This should be done to not fatigue the user with items of similar categories. An example would be the recommendation of movies from a variety of different genres.

Throughout the years, different approaches have been developed to optimize recommendations and overcome obstacles in this field. These recommendation systems are broadly categorized into content-based RSs, collaborative filtering (CF), knowledge-based RSs and hybrid RSs (Aggarwal, 2016). In this thesis, CF will be described, as it is a key concept for the RS implemented in the artifact. Figure 1 shows the structure and an overview of the recommenders based on Aggarwal (2016). Highlighted in red is Google's novel RS proposal, the SotA approach covered in this work. Before describing CF, there will be a chapter dedicated to retrieval and ranking, two tasks RSs are used for, followed by a chapter about RS data.

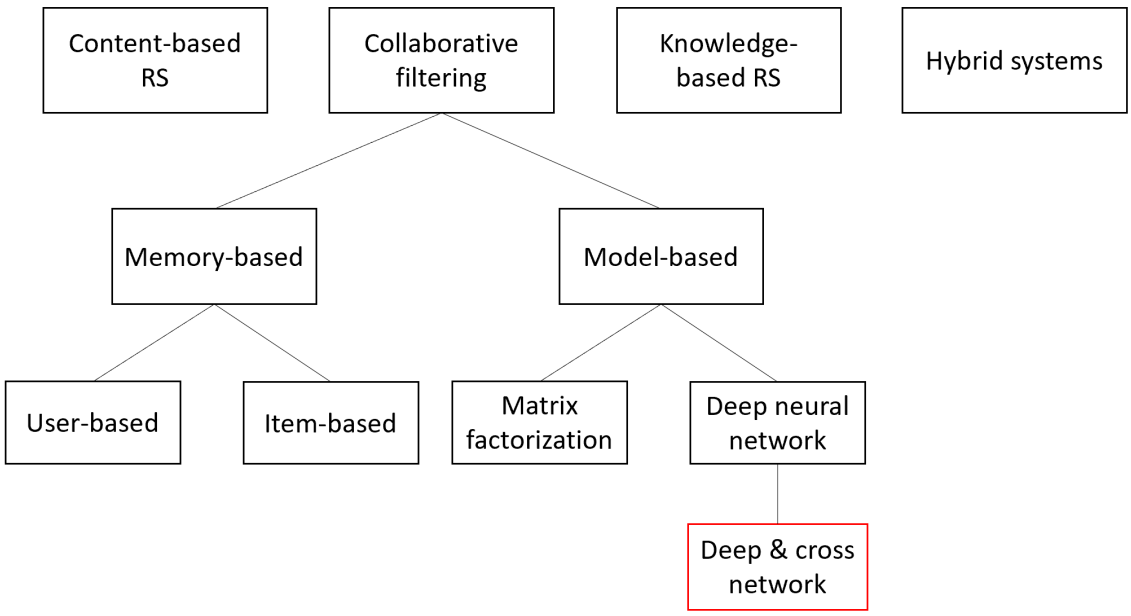


Figure 1: structure and overview of RSs derived from Aggarwal (2016)

2.2.2 Retrieval & Ranking

Most RS algorithms are employed to fulfill either a retrieval or a ranking task. The retrieval task deals with the generation of appropriate candidates for a recommendation.

Within a dataset, a subset of similar items and users are identified (Fernández-Tobias, Cantador, Kaminskas, & Ricci, 2012). The objective is to narrow down the dataset to the relevant users and items that are then used by a ranking algorithm to make scoring predictions. Opposed to the retrieval task, the ranking task makes concrete recommendations to a user. Ranking can manifest itself in form of a predictive score, or a top-items list (Google LLC, 2020b, 2020c).

Neighborhood algorithms and other similarity measures are popular methods used for retrieval operations, whereas SotA ranking is often done via matrix factorization or deep learning models. Neighborhood algorithms, like k-nearest-neighbors, can be used for ranking as well, as explained in chapter Collaborative filtering.

In practice, both retrieval and ranking are often combined into a joint RS, as seen in Figure 2. Since ranking algorithms are computationally more expensive, they wouldn't scale well in an enterprise setting. For this, the faster retrieval operation pre-selects a set of candidates, which are then run against the ranking model.

Retrieval systems are outside the scope of this research project and thus won't be further discussed. All references about RS in this paper are related to ranking models.

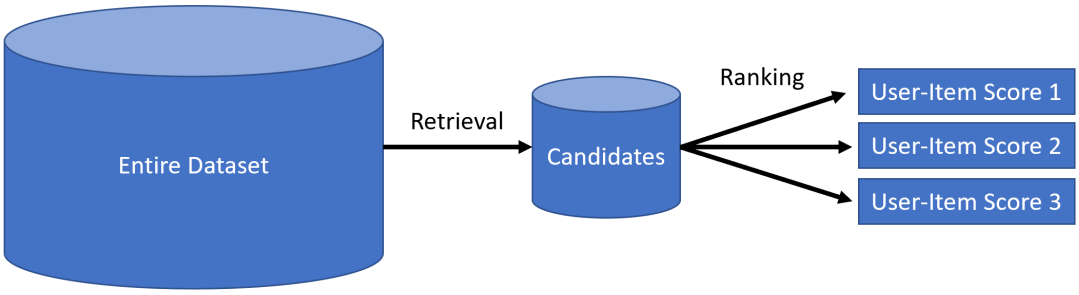


Figure 2: Example of retrieval and ranking in a recommender system

2.2.3 Data

The data used by RSs often differ from those of other ML tasks, as it is often sparse. Sparsity refers to very few user-item interactions relative to the size of the whole dataset. Assuming one would construct a $U \times V^T$ matrix of an e-commerce website, wherein U represents the entirety of its user-base and V represents all the items in the product catalogue. Every time a customer buys or inspects a product, an interaction between the user and the product is stored in (u_i, v_j). Since customers only interact with a fraction of the offered products in an e-commerce web shop, very few entries in the 2-dimensional matrix are stored. Such matrix is called sparse. This is a fundamental problem that RSs face and try to overcome.

The value of an interaction between item and user can be represented in different ways depending on the context of the recommendation. Is an interaction defined as a review with a rating between 1 and 5 stars, it will be marked with an integer value in a range of

1 to 5. These rating types allow the user to quantifiably express their like or dislike of an item. These are called interval-based ratings (Aggarwal, 2016). Another popular rating type is the unary rating system. These are used on social media platforms like Instagram and Twitter. These ratings are encoded in a binary fashion, wherein an interaction (e.g. like) will be conventionally denoted with the value 1 and abstinence will be denoted by the value 0 (Aggarwal, 2016).

Both the like-feature as well as the 5-star rating system are called explicit ratings, as they are actively and consciously given by a user. While explicit ratings generally provide reliable information about an user's opinion, it remains a challenge to get users to rate. This in return makes the data more susceptible to shilling attacks, where item ratings get skewed by coordinated bombardments of overly positive or negative ratings by external sources (Khusro et al., 2016).

In opposition to explicit ratings, lie the implicit ratings. These rating systems derive a rating from the user through their behavior. An example could be the watch time of a movie. This example works under the assumption that people that don't close a movie, are engaged with it. The drawback of an implicit rating is its ambiguity, as online behavior needs to be interpreted first and then translated into a rating. User behavior can be interpreted wrongly and skew the data. Continuing the aforementioned movie example, a user who slept during the movie out of boredom, would be misinterpreted as a user who was engaged throughout the whole movie, since they didn't close it. An implicit rating system has the advantage that it is not reliant on the user to give feedback, which generally results in more ratings. This is a significant factor considering sparsity is a main challenge for RSs (Su & Khoshgoftaar, 2009).

In a recommender dataset the value of a user-item interaction is referred to as a *sparse feature*. They can be augmented with so-called *dense features*. Whereas sparse features only occur through user-item interaction (e.g., a rating), dense features are specific attributes that describe the item or user at hand, comparable to features in other ML tasks.

Data understanding plays a vital part in the success of RSs. There are many variables ranging from its rating system over its features and sparsity that determine, which RS is the most suitable for the given task (Aggarwal, 2016, p. 128).

2.2.4 Collaborative filtering

CF has been the baseline for RSs for years. In September 2009, the BellKor team was awarded with the grand Netflix prize, which it has won by applying a CF model on the competition dataset (Koren, 2009a). As the name implies, CF uses the "collaborative" power of the user-item interactions in the dataset (Aggarwal, 2016). Instead of solely relying on item and user descriptions, CF uses the composition of the dataset to determine the output of a recommendation. Colloquially, one would say that the recommendation is driven by a "joint effort" of every user-item interaction. Unlike in content-

based methods, recommendations can therefore be made without relying on hand-engineered features (Google LLC, 2021a).

CF can be divided into *memory-based* methods and *model-based* methods. The former methods predominantly use neighborhood-based algorithms on the item-user interaction to make recommendations whereas model-based methods have a training phase to first establish a model describing these interactions. In the later method, the goal is to create a function (i.e., the model) that approximates the results of an item-user interaction, which is conventionally achieved through optimization of a loss function with gradient descent or alternating least square algorithms.

As model-based methods represent the SotA for RSs, for this research we will only cover model-based RS methods. Two of the most prominent model-based CF techniques are matrix factorization (MF) and deep learning (DL), which will be described here.

Matrix Factorization. This has been the standard in the field of RS and was also utilized by the winning team of the Netflix prize in 2009 (Koren, 2009a). MF is foremost a dimensionality reduction technique with which the user-item matrix gets decomposed into *low-rank*[2] *latent factors*. These latent factors are the components that make up the *latent factor model* (LFM), which is a low dimensional representation of the initial matrix. LFM, broadly classified as a *factorization machine* (FM), works under the assumption that there are underlying latent variables in the data. Latent variables are variables that are not directly observable but can be inferred through mathematical computation, like for example singular value decomposition (SVD) (Loehlin & Beaujean, 2017). Koren et al. (2022) state that the observed rating values are due to effects associated with either user or items, independently of their joint interaction. Therefore, there are large item and user biases embedded in the data. An example would be the tendency for some users to give either higher or lower ratings on average, or conversely products that receive systematically higher or lower ratings. These “hidden” factors can be retrieved during model training and incorporated into a latent factor. In the training process the latent factors are distilled that best depict these propensities in the data (Vellido, Lisboa, & Meehan, 2000). In LFM, the rating of an unobserved user-item interaction is constructed by applying the dot product of factor of user u_i and factor of item v_j (Aggarwal, 2016). While the baseline LFMs only use the sparse matrix of user-item interactions, more sophisticated variants, like SVD++, can also incorporate dense features and implicit ratings into its model, which allows it to factorize more information and further improve prediction quality (Koren et al., 2009).

The main advantage of MF lies in its ability to algorithmically detect latent vectors in the dataset. Overarching correlations are extracted in the dataset and used to make predictions. Unlike CB methods, the models don’t rely on hand-engineered features to

[2] Rank refers to the dimensionality k of a latent factor $m \times k$ for Matrix U and $n \times k$ for Matrix V, low-rank meaning that $k \ll \min\{m, n\}$. k represents the amount of linearly independent factors in a latent factor (Aggarwal (2016).

make recommendations, which evades the human-error component of feature engineering and speeds up the development process. Dot products are computationally light, which makes LFM models scalable and applicable on large datasets (Blondel, Ishihata, Fujino, & Ueda, 2016). However, the performance advantage of FMs rapidly diminish when modeling higher-order feature combinations, which is a tradeoff to allow for more complex embeddings in the model (Blondel, Fujino, Ueda, & Ishihata, 2016). FMs being bound by their shallow structure hinders their overall representative power (Wang, Fu, Fu, & Wang, 2017). In practice, simple FMs still are very capable at making accurate predictions (Dacrema, Boglio, Cremonesi, & Jannach, 2021).

Deep Learning. In order to circumvent the above-mentioned limitations of FMs, *deep neural networks* (DNN) have gained increasing interest in RS research over the past years (Dacrema et al., 2021). *Neural networks* (NN) are very effective at retrieving complex embedded information from their input, which makes them state of the art in the field of image recognition and natural language processing.

NNs emulate a simplified model of how human neurons actually work in a brain. Like the brain in biology, *artificial neural networks*[3] (ANN) consist of multiple individual neurons that are interconnected, whose joint response to an external input creates an output. Unlike neurons in our brains, which are structured in a complex and organic way, neural networks are arranged in layers, which contain a set of artificial neurons (Charniak, 2019).

The baseline neuron in NNs can be thought of as an individual mathematical function, that consists of a weight component and a bias component, which are wrapped inside an activation function. This neural function receives an input, which can either be the external input data, or the output of neurons from the previous layer. In the neuron, the input value gets multiplied by the weight, then the bias offsets the product by a certain value. The activation function refactors this weighted sum, which makes the output value of the neuron. Activation functions are non-linear functions that determine how "activated" a neuron is, i.e., how large the output value should be. A popular activation function is *rectified linear unit* (ReLU), which represses any activations (i.e., neuron output value of 0) up to a certain threshold (Nair & Hinton, 2010). For the output layer (the last layer of the network) activation functions like *Softmax*[4] are preferred, as they normalize the neural computation to a probability distribution. For classification tasks, an *Argmax* function can be applied on the output layer to retrieve the neuron with the highest Softmax output, which represent the class the NN predicts with the highest confidence (GOODFELLOW, BENGIO, & COURVILLE, 2016). The non-linearity of these activation functions in combination with the multiplicity of artificial neurons, allow NNs to retrieve highly non-linear patterns from the data, which is the reason they are

[3] NNs and ANNs will be used interchangeably in this paper.

[4] The Softmax function maps the neural output on range [0;1]. Highly negative inputs approach value 0, while highly positive inputs approach value 1.

also referred to as universal function approximators (Charniak, 2019; Gurney, 2014; Hanin, 2019; Heaton, 2012).

NNs are learned through *deep learning*. Intuitively speaking, the objective of the learning process is to tweak each parameter (i.e., weight and bias) in a NN in such a way that a desired output corresponding to a certain input is generated. The training is done through an iterative process called *gradient descent*. Before the training process begins, each weight in the network gets assigned an initial random value. This offsets the NN to learn distinct embeddings, which would not be possible if each weight received the same value. During the training process *test data* is fed through the network. With a loss function the desired target output y is compared to the actual output of the unoptimized model $\hat{y}$. The goal with each training iteration is to decrease the distance between $\hat{y}$ and y. To achieve this, the gradient vector of the loss function is calculated with respect to the entire NN. This computation of the gradient in a NN is what's called *backpropagation*. Earlier it was established, that each neuron is a function of the outputs from the neurons in the previous layer. Backpropagation recursively calculates the gradient inside each neuron in the NN, starting with the output layer and propagating through the entire network to the input layer. Having calculated the gradient, the opposite sign of it is taken and multiplied by the *step size*, a coefficient which defines how aggressively the parameter values are tweaked. This process is repeated until the NN function converges to a local minimum. A small step size increases the learning time, since it takes longer to reach a minimum. Conversely, a large step-size runs the risk of exceeding the local minimum, which results in an inferior model (Gurney, 2014; Heaton, 2012; LeCun et al., 1989).

For every model training, it is crucial to separate the training data from the test data to prevent an overfit of the model. This is especially true for NNs, as they have the tendency to perfectly adapt to the data it gets trained with, which greatly impacts the generalizability of the model and impairs prediction performance on unseen data (May, Maier, & Dandy, 2010). In general, the learning of NNs is a non-trivial task, as they are very sensible to the embedding of the input data, the initialization of the weights, the choice of activation functions and hyperparameter tuning, like step size and *dropout regularization*[5]. A lot of variables that determine the efficacy of a model make it harder to extract the full potential out of a NN. In addition, NNs can't easily be interpreted because of their complexity, which is why they are often treated as black boxes, especially with very deep models. Dacrema et al. (2021) discovered that simple MF methods show similar performance to other SotA DL models, when selecting the right parameters. Compared to a linear LFM, NNs also have a lower processing speed, which is detrimental to large recommendation datasets, since passing a neural network is much more computationally expensive than a simple dot product calculation (Rendle, Krichene, Zhang, & Anderson, 2020).

[5] In dropout regularization random neurons are skipped during the training process to combat overfit of the model (Hinton, Srivastava, Krizhevsky, Sutskever, and Salakhutdinov (2012).

This notion changes when comparing NNs to polynomial FM, which are much more computationally complex compared to their low-degree counterparts (Blondel, Fujino, et al., 2016). As universal function approximators, DNNs have the potential to extract much more complex patterns and relationships out of the dataset, especially as both dense and sparse features can be trivially incorporated into its embedding. With ongoing research, DNN approaches for RSs are further being refined and its deficiencies, like performance, investigated and improved. Currently, high profile enterprises like Google use NNs in large scale environments for recommendation tasks (Covington, Adams, & Sargin, 2016).

2.2.5 Deep & Cross Networks

When using data with multiple features to make recommendations, *feature crosses* become an important concept. The notion of feature crosses is that the combination of multiple features into one feature can yield an expressive predictor variable, that would otherwise not be extractable using only single features. Individual features that have very low correlation with the label can therefore reveal themselves to be highly correlative in conjunction with other features. Uncovering these *cross features* allows to extract additional information out of the data, which in return can significantly improve model performance. The number of features that are embedded in a cross feature is denoted by the order: A cross feature being the product of two features is called a cross feature of 2nd order. Cross features of 3rd order and up are referred to as higher-order features (Wang et al., 2017; Wang et al., 2021).

Suppose an RS that predicts the user rating of a movie. Besides the sparse features (in this case the user ID and movie ID), the data also contains the genre and length of the movie as dense features. Assuming that length as a stand-alone feature is not a strong predictor for the rating of a movie, it could be combined with the genre feature into a separate cross feature of 2nd order. Now, a feature that has both variables encoded in it can be used to better predict the ratings of a movie, the hypothetical interpretation being, that different genres set certain expectations of what the movie length should be. For instance, while it might be acceptable or even desired for fantasy movies to be 3 hours long, a 3-hour runtime for a comedy could generally be viewed unfavorably.

Feature crossings are often done as part the data preparation phase of a ML lifecycle (Studer et al., 2021). This process can be very daunting and time consuming, as it entails trial and error of which cross features are best suited for accurate predictions. Also, feature crosses aren’t always obvious and thus might go unnoticed. Instead of manually feature engineering cross interactions, it would be lucrative to directly incorporate feature crosses into the ML model and let the learning algorithm determine the weights of cross features. DNNs are capable of learning arbitrarily high order functions, provided the NN is deep enough. However, they only learn feature interactions implicitly, which means that they do not reliably pick up on higher feature crosses without drastically increasing the network size (Wang et al., 2017). This brings the general effi-

ciency of DNNs into question, especially in light of large-scale recommendations that need to compute high amounts of queries per second (Rendle et al., 2020).

In order to reduce the overall model size of NNs and invoke explicit cross feature learning, Google proposes their deep and cross network (DCN) in 2017 (Wang et al., 2017). In addition to a classic feed-forward multilayer perceptron (MLP) NN, as covered previously, a DCN is augmented by an additional cross network, consisting of cross layers. The cross network enables learning on bounded-degree cross features, whose order is explicitly defined by the layer depth of the cross network. This way, ML engineers can now directly incorporate the order of feature interactions they wish to embed, by changing the cross network structure. At the start of the network (i.e., the input layer) the feature vector x_0 doesn't have any explicit cross feature interactions yet and is therefore of order 1. With each cross layer the feature vector passes, the maximum polynomial degree of cross interactions increases by 1. This is because for each cross layer the output of the previous layer x_i gets crossed again with the input vector x_0 to construct a new cross term x_{i+1}. A visualization of a cross layer in Google's DCN V2 architecture is seen in Figure 3, whereby the learned parameters W and b respectively denote the weight matrix and the bias of the cross layer.

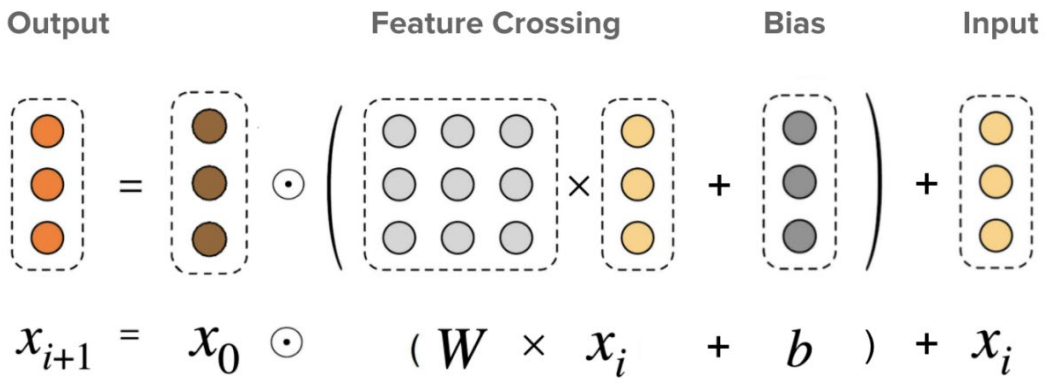


Figure 3: Visualization of a cross layer (Wang et al., 2021)

A DCN is always made up of two components: A cross network and a conventional MLP neural network. While the cross network is used to learn explicit bounded-degree feature interactions, the DNN component is used to learn implicit feature interactions in the data[6]. With both components the model is therefore able to effectively leverage explicit and implicit interactions in the data, which improves prediction performance.

[6] Explicit feature interactions are usually retrieved by dedicated components in the model, such as the cross network in DCN. The order of feature interaction can be set (bounded-degree) and inferred by said component, which makes the feature extraction more controllable and predictable. On the contrary, implicit feature interactions can have an arbitrary order, as it is not explicitly defined in the model to which degree the interaction is bound to (i.e. DNN). Implicit feature extraction is more flexible at picking up any patterns in the data, at the cost of certainty of the degree of the feature interaction (Lian et al. (2018); Huang, She, Wang, and Zhang (2020); Yan and Li (2020).

Depending on what architecture is chosen for the DCN, the DNN can either follow the cross network, or run in parallel, as seen in Figure 4.

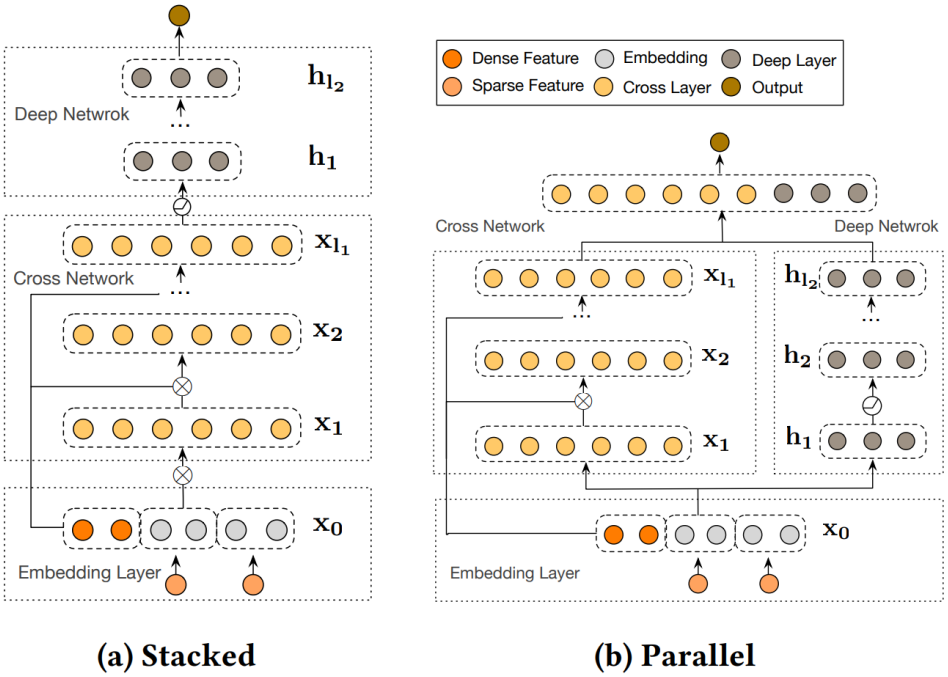


Figure 4: Possible DCN architectures (Wang et al., 2021)

With the introduction of cross layers, the model becomes more effective and predictable at embedding feature interactions into its network. With DCNs, the task of finding cross feature interactions can now be delegated to the model training (Shan et al., 2016). This saves labor time and diminishes the human error element. Being able to extract both implicit and explicit feature interactions in the data through the combination of a cross network and DNN, makes DCN overall more competitive at predicting ratings. Besides improving the prediction performance of recommendations, the model also becomes smaller and in return faster, which makes it more viable for large-scale RS operations.

2.2.6 State of the Art Technology

TensorFlow Recommenders (TFRS) is a python library built on the Keras API. TFRS unifies and simplifies the building of DL RS models under TensorFlow, by providing functionality and components at each step of the RS development cycle, such as methods for retrieving and ranking recommendations, activations functions, and ready-to-use RS datasets (TensorFlow, n.d.). TFRS also provides built-in RS specific network layers for model building, such as the aforementioned cross layers. In combination with the accessibility of the Keras API, it opens the possibility to intuitively realize SotA DCNs (TensorFlow, 2022a, 2022b).

2.3 MLOps

2.3.1 Problem

ML has its origins in academic research (Minsky, 1961; Rosenblatt, 1961). Over the last two decades ML also found its way into the enterprise sector, where it now forms the backbone of the most successful companies and most profitable business ventures. With the transition from academia to industry, ML engineers and data scientists are confronted with unique sets of challenges that are usually absent in a sterile research environment. While research often focusses on one aspect of ML (e.g., the algorithm), enterprises devote themselves to the development, operation, and maintenance of entire ML systems. Software in research is written to investigate a certain topic and to accept or decline a very specific hypothesis. It usually has a very narrow use-case and therefore lacks generalization, which makes it often not suitable for a productive setting. In addition, development of the software usually stops with the conclusion of the research project (Garousi, Petersen, & Ozkan, 2016). On the other hand, software employed in business environments are built for longevity, with their goal being a profit driver for the company. As a result of these different goals, both research and industry employ different development paradigms, with focus on different aspects respectively. These differences are observed in the RS landscape of academia and industry, for example. While in academia SotA RS algorithms are assessed with their performance on static benchmark datasets (e.g., MovieLens dataset), industry also considers the integration of the algorithm into the system architecture. Instead of merely relying on benchmark performance, industry systems need to take into account the scalability of an algorithm and embed their RS into their application in a meaningful way (Covington et al., 2016).

One of the first papers to illustrate challenges of productive ML systems is “Hidden technical debt in Machine Learning Systems” from Sculley et al. (2015), which uses the framework of *technical debt*, found in software engineering, to describe problems in a productive software environment. To be more precise, the term technical debt refers to ongoing maintenance cost due to shortcuts taken during initial development of the software. The debt analogy is used to visualize upfront cost-savings due to cheaper and faster development, that however need to be paid back with interest down the line with excessive software service and maintenance (Cunningham, 1993). In classical software engineering, technical debt can manifest itself in different ways, such as code refactoring, deletion of outdated modules, reduction of dependencies and improvement of the documentation (Fowler, 2019). The same phenomenon of technical debt is also observed in ML systems.

Sculley et al. (2015) points out that a lot of the technical debt incurred in ML, is due to the fact that it directly interacts with the external world. Since ML holds a tight bond with the ever-changing data it consumes in real-world applications, maintenance turns out to be particularly difficult and expensive. So, in addition to the established technical debt of software engineering, ML also faces its own set of technical debt that needs to

be considered (Sculley et al., 2015). Not acknowledging changes in data can lead to an amalgamation of technical debt, to the point where it might render the whole model obsolete (Lu et al., 2018; Sculley et al., 2015). This problem is dissected in greater detail in chapter Concept Drift.

Enterprise ML systems are often complex and comprise various interconnected components. Consequently, the ML algorithm itself only makes up for a small portion of the entire system. In addition to the complex array of variables in the ML model alone, all parameters from the other software components effect the ML system as well, which results in interdependence. It becomes hard to keep track of all dependencies within the system, which, in return, complicates the maintenance of the software, which is referred to as *configuration debt*. The *"changing anything changes everything"* principle goes into effect and any adjustments to the system can severely compromise the model performance (Sculley et al., 2015). Related to the configuration debt, is the *reproducibility debt* in ML systems. Reproducibility is an inherently difficult task in productive ML systems, as these algorithms often work with randomized parameters, non-determinism in parallel learning and continuously changing data.

2.3.2 What is MLOps

Paying down ML related technical debt demands a re-examination of status-quo practices in a business environment. Industry has different sets of priorities compared to academia, which need to be considered. Instead of determining the viability of a model based on specific performance metrics alone, the effects on operability of the whole system need to be taken into account. Small performance gains at the cost of maintainability usually are not recommended in an enterprise environment. At the same time, infrastructure needs to be set in place, that can prevent and mitigate technical debt. This entails sustainable design patterns in ML code, such as modularity and abstraction, rigorous testing, as well as practices that promote team work (Sculley et al., 2015). The goal for the last few years, was to apply practices of agile software development to ML. As assessed in the prior chapter, ML development contains a few peculiarities, that sets it apart from conventional software development, which makes it incompatible with common technical methodologies, like DevOps. Out of necessity to employ methodologies that are attuned to the unique characteristics of ML, the term MLOps was born. Despite gaining attention from high profile ML researchers and companies, MLOps still remains a vague term that is hard to pin down (Kreuzberger, Kühl, & Hirschl, 2022; Tamburri, 2020). This is due to the novelty of this field, with different sources defining MLOps differently.

A Google whitepaper about this topic defines MLOps as "*a methodology for ML engineering that unifies ML system development (the ML element) with ML system operations (the Ops element)*" (Salama, Kazmierczak, & Schut, 2021, p. 5). Similar to DevOps, MLOps seeks to employ practices and tools to shorten development cycles and enable rapid iterations of ML software (Symeonidis, Nerantzis, Kazakis, & Papakostas, 2022). Streamlined operational and governance processes are used to increase

performance, reliability and security of ML systems. To sustain this agile development, focus is put on collaboration methods and tools. One overarching theme of MLOps is the unification of research, which is inherent to ML, with the efficiency of modern software development to make ML more profitable and scalable in real world applications (Salama et al., 2021; Sculley et al., 2015).

Tamburri (2020) defines MLOps from a technological perspective as a set of middleware and software components, that ought to be orchestrated in the cloud to realize at minimum 5 specific functions:

- Data ingestion & transport
- Data transformation
- Continuous ML (re-)training
- Continuous ML (re-)deployment
- Output production & presentation to the end-user

These points are derived from popular cloud MLOps platforms, like Kubeflow, which enable scalable ML pipeline orchestration through containerization (Kubeflow, n.d.). These platforms provide a large portion of the infrastructure to realize SotA productive ML systems, that can be broken down into following areas of responsibility, which are automation, reproducibility and monitoring:

Automation. Due to changing data that ML systems consume, they need to be constantly updated and deployed. Employing a CI/CD (continuous integration/continuous delivery) approach for ML, allows development teams to quickly release updated iterations of models and other components. In CI/CD, new models can be automatically compiled into a build and be deployed instantaneously in the needed environment.

CI/CD is often realized in conjunction with automated pipelines, which are a vital part of successful ML operations. A typical ML lifecycle usually consists of repeating steps. Taking the CRISP-ML process as an example, every development cycle needs to run through a data preparation phase (Studer et al., 2021). Since these steps remains consistent throughout different lifecycles, they ought to be automated. Pipelines remove manual labor from the ML process, which speeds up the development of new models, reduces the human error element and makes the process more consistent.

In SotA MLOps, the conventional CI/CD practice is expanded by *continuous training*. CT aims to automate the entire model training process. The ML system should be able to detect deteriorating model performance and initiate a model retraining pipeline (Denis Baylor et al., 2019). One form of achieving CT is deploying entire ML pipelines instead of a trained model. This way, instead of deploying a single ML model at the time, an entire pipeline can be deployed to recurrently create new and optimized models (Denis Baylor et al., 2019).

Reproducibility. The most straight-forward implementation of reproducibility is through versioning of the source code with a *version control system* (VCS), like Git. As an industry standard, VCSs allow to trace and revert changes in the code. This way previ-

ous states of the software code can be restored and rebuilt (Serban, van der Blom, Hoos, & Visser, 2020).

However, for ML applications, code versioning alone is not sufficient. Since model training is not only dependent on the model configuration, but also the training data, the dataset needs to be versioned and tracked as well, in order to replicate a previous model (Ruf, Madan, Reich, & Ould-Abdeslam, 2021). This can be achieved through time travel functionality of modern data lakehouse architectures, such as Databricks delta lake (Yavuz & Chockalingam, 2019). Alternatively there are dedicated data management tools, like Rok from Arrikto, integrated into Kubeflow (Arrikto, n.d.).

As alluded to earlier, ML utilizes a wide array of hyperparameters and randomization-seeds for learning. These need to be tracked as well, in order to ensure reproducibility of the model. These are stored in a meta data store, alongside other information, such as training duration and training date.

Monitoring. Continuous monitoring is used in MLOps to assess the health of deployed ML systems. Most importantly, the prediction quality of served models are frequently measured in an automated fashion. Important metrics are given a threshold value, that are tracked live by the monitoring system. Sophisticated monitoring implementations automatically trigger pipelines to take corrective actions, upon noticing insufficient model performance (e.g., a re-training pipeline). The observations made during monitoring can feed back into the consecutive model to make it more robust. Automated monitoring can be supported by human monitoring through dashboard visualizations on applications like Kibana, Grafana, or PowerBI (Kreuzberger et al., 2022).

2.3.3 Pipelines

Pipelines are an important component in the operationalization and automation of ML Lifecycles. Generally, processes in information systems can be explained with graphs. Graphs are the subject of graph theory, a discipline in mathematics and computer science. They are made up of two components: nodes and edges. Edges are used to map relationships between nodes. Graphs can define dependencies between individual tasks of a process, which make up a pipeline. Specifically, in the context of Pipelines, a special type of graph is used. The *directed acyclic graph* (DAG). A DAG has directional edges, which establish a chronological order between the nodes. Instead of only mapping a generalized relationship, directed edges represent a clear dependency of one node to another node. These dependencies can for instance constitute a data flow. “Acyclic” means that edges can’t connect to prior nodes to form a cycle or loop. The acyclic properties of a DAG are important for ISs. Circular dependencies can provoke deadlocks in a pipeline, as nodes can now be dependent on multiple nodes (Deo, 2019; Harenslak & Ruiter, 2021).

DAGs can be created and interpreted by workflow management tools like the Apache Airflow or Apache Beam (Apache Software Foundation, n.d.). These tools execute individual work steps in the defined order and logic of the graph. An exemplary DAG in

Airflow is shown in Figure 5. In order for a pipeline to run, it needs to be triggered by an external event. This can be done with event listeners or pipeline schedules.

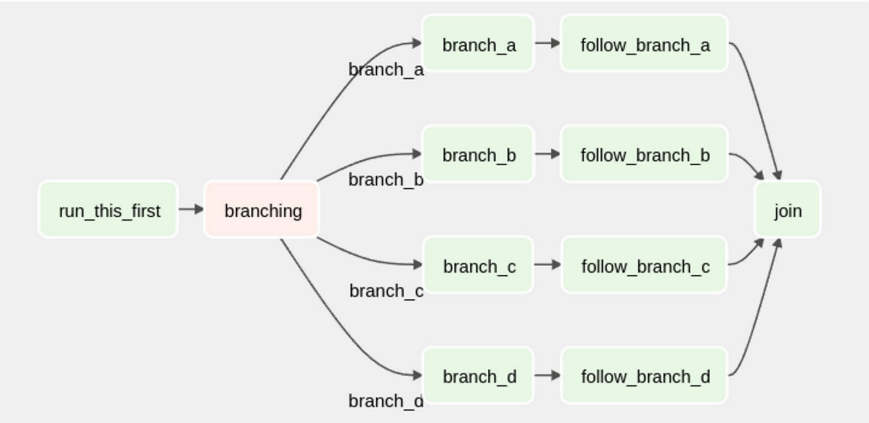


Figure 5: Exemplary DAG in Airflow

With pipelines, previously manual tasks can be automated by executing individual scripts represented by a node. Airflow and Beam can be used to orchestrate the data engineering steps, for instance. In the grander scheme of things, the industry is developing workflow solution that are designed to map a whole end-to-end ML lifecycle process in a structured way. Some of these tools are discussed in chapter *State of the Art Technology*. Because of the deterministic nature of DAGs, they can also ensure reproducibility in the ML lifecycle. Pipelines promote compartmentalization of software components, which in return promote modularity and maintainability in software systems. This is especially beneficial in ML systems, which are often composed of multiple individual and interconnected tasks. Pipelines are essential to the MLOps methodology, as they streamline the development of ML models and ML systems. With the use of pipelines, developers are able to release software in a more frequent fashion (Alla & Adari, 2021; Hutter, Kotthoff, & Vanschoren, 2019).

2.3.4 Maturity Levels

Maturity levels are used to determine the sophistication of an MLOps implementation. The maturity of a ML System can be defined by its technological infrastructure, employed methods of operation, and automation through pipeline orchestration. Because of the novelty of the field of MLOps, there isn't one universal maturity model agreed upon by the scientific community. Companies that are heavily invested in AI therefore design and develop their own maturity models. The most prolific proposals stem from Microsoft and Google (Symeonidis et al., 2022). In the following, Google’s maturity model is described.

Google’s maturity model is divided into 3 levels. Level 0: Manual process, Level 1: ML pipeline automation, and Level 3: CI/CD pipeline automation. A higher level means a more sophisticated implementation of MLOps for the ML system (Google LLC, 2020a).

Level 0 denotes the baseline maturity, where no MLOps practices are employed. Every step is done manually, usually executed interactively on a notebook. There is no distinction between experimental and productive code, as iterations of the model are done on the same notebook until a suitable model is trained. No CI/CD is employed, which means that testing is usually done in an ad-hoc manner within the notebook. Once the model is deployed there isn't a monitoring infrastructure to observe the model performance in production. Consequently, level 0 ML systems don't deploy models regularly and run the risk of not noticing degrading prediction quality.

The goal of level 1 ML systems is to perform continuous training. In level 0 a trained model is deployed. In level 1 a complete ML pipeline is deployed, which triggers a model training process. The pipeline then continuously delivers up-to-date models that are then used by the application. In production, the data and the model are monitored to detect potential shifts and skews, which then can trigger the pipeline to retrain the model. In order to improve reusability and accelerate the pipeline development process, the source code is broken up into modular components. There is now a strict separation between development and production environment.

In Level 2 ML systems, CI/CD is employed to the full extent. The test and deployment of pipelines are automated and the experimentation process is orchestrated. Intelligent monitoring is used on live data, which is able to discern whether a model retraining or a new experiment cycle should be initiated. Information about models, data and metadata are stored in a model registry, feature store[7] and metadata store. This makes ML pipelines reproducible and transparent.

2.3.5 Concept Drift

"*One of the things that makes ML systems so fascinating is that they often interact directly with the external world. Experience has shown that the external world is rarely stable. This background rate of change creates ongoing maintenance cost.*" (Sculley et al., 2015)

One manifestation of ongoing change in data is called *concept drift*. Specifically, CD describes a changing outcome y to a constant input X over time (Ng, Crowe, & Moroney, 2021). Real world examples of CD could be changing house prices due to a fluctuating house market, or people changing their taste in movies because of aging or genre trends. These changes in the underlying distribution of ML data are often unforeseeable. Therefore, they are hard to detect and often go unnoticed for a long time. CD runs the risk of first being noticed through deterioration of model performance in a productive environment.

[7] A feature store is a data management layer that tracks and maintains features in ML data. Feature stores simplify ML pipelines and the data engineering process, by providing a centralized space where all raw and transformed data features are stored in. Previous data transformation steps are saved in the feature store, where they can be reused by other data scientists (Kakantousis et al. (2019).

Gama et al. (2014) identify 4 common types of CD as seen in Figure 7:

Sudden drift. The distribution[8] of the data undergoes an abrupt change in a short amount of time that persists indefinitely. This phenomenon is also known as *concept shift*. One example for concept shift would be the drastic drop in demand for commercial flight to a country where war just broke out (International Air Transport Association, 2022).

Gradual drift. This type of concept drift has two active concepts in the data. Over time, the probability of the initial distribution decreases and gets gradually displaced by the new distribution. An example could be the genre representation in the billboard top 100 charts. As new genre trends emerge, old ones fade out and get replaced by the new ones. During the transitory period both concepts coexist in the data.

Incremental drift. Unlike gradual drift, the incremental drift can't reliably be ascribed to a discrete number of distributions. The drift occurs slowly with a lot of variables and concepts at play. Consequently, incremental drift is only noticeable when looked at over a long period. Cultural and societal changes are often incremental in nature.

Reoccurring concepts. Previously seen distributions might reappear after some time, as an irregular occurrence. While seasonal changes are reoccurring, they wouldn't necessarily be classified as concept drift, as they are predictable to an extent. An example for reoccurring concepts are rare weather phenomena like hurricanes (Knotek & Pereira, 2011).

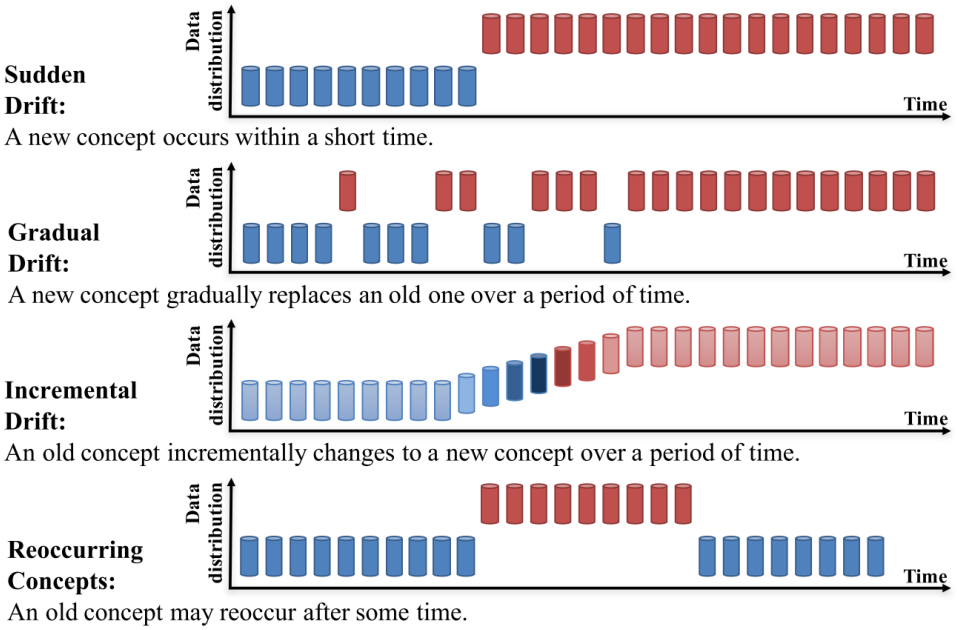


Figure 6: Types of concept drift (Lu et al., 2018)

[8] Distribution and concept are used synonymously in the context of concept drift.

Not all changes in data can be attributed to CD. A major challenge in CD detection is to differentiate true CD from noise and outliers. Noise and outliers are random deviations in the data, that aren't indicative of any larger patterns and shouldn't therefore be treated as CD (Gama et al., 2014).

Throughout the years, researchers have developed frameworks and algorithms to deal with CD. Lu et al. (2018) proposes a framework to integrate CD handling into a production ML system (Figure 8). In this proposal CD handling consists of 3 parts: CD detection, CD understanding, and CD adaptation.

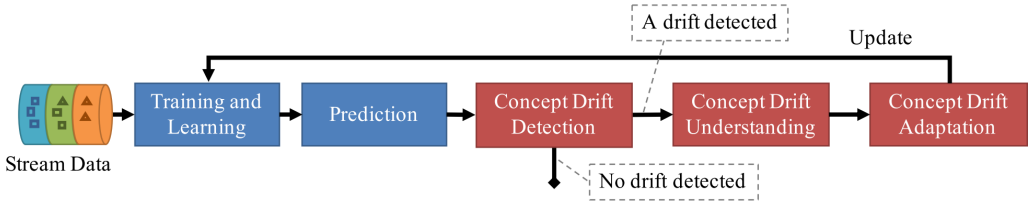


Figure 7: Framework for handling concept drift in machine learning (Lu et al., 2018)

CD detection. CD's unpredictable and inconspicuous nature makes it often hard to identify. CD detection employs techniques and mechanisms to characterize and quantify CD. CD can be uncovered in different ways.

The most common way to detect CD is through observation of model performance in the productive environment. This type of CD detection is referred to as *error rate-based drift detection*. A performance decrease over time is indicative that characteristics of the data have shifted. Different algorithms can be employed to analyze CD using the model performance metrics. Drift Detection Method (DDM) was the first algorithm to assess the severity of the CD at hand, based on how the model performed over a set time window (Gama, Medas, Castillo, & Rodrigues, 2004). The window defines the timeframe of past model performance, which the current model performance is compared against. If DDM detects a significant error-rate increase, compared to the old data, it'll declare that CD has occurred. Built upon DMM, other algorithms have been developed, such as the Early Drift Detection Method (EDDM) and the Fuzzy Windowing Drift Detection Method (FW-DDM) (Baena-Garcıa et al., 2006; Liu, Zhang, & Lu, 2017). In the context of automated CT pipelines, it becomes especially important to configure appropriate values for CD detection. The right performance metrics need to be monitored and the right performance thresholds need to be selected. Additionally, the performance needs to be observed over the right time window. Are these values not set correctly, CD detection might become too sensitive to noise. Conversely, the CD detection might pick up on CD too late.

Instead of using model performance to determine CD, the dataset can be analyzed. This category of algorithms is called *data distribution-based drift detection*. In this case, severity is quantified using distance functions to measure dissimilarities between new and old data (Lu et al., 2018). Density based algorithms and Principle Components

Analysis (PCA)-based algorithms can be employed for CD detection (Feng Gu, Zhang, Jie Lu, & Chin-Teng Lin, 2016; Qahtan, Alharbi, Wang, & Zhang, 2015).

CD understanding. CD understanding deals with the extraction of important information about the CD at hand. CD understanding usually is the output of CD detection algorithms. Additional understanding can be acquired by employing additional methods, like visualization. According to Lu et al. (2018), following questions should be answered in order to gain a comprehensive CD understanding:

- When did CD occur and how long does it last? (When)
- How severe is the concept drift? (How severe)
- What were the drift regions of CD? (Where)

Identifying when CD occurs is the baseline requirement for every CD detection method and therefore the most trivial question to answer. Detecting CD implicitly contains information about when the CD at hand was detected. Through continuous monitoring of the data and model, the duration of CD can be measured as well.

In order to satisfy the second question, CD needs to be quantified in a non-binary fashion. The severity of CD is measured by most dedicated CD detection algorithms, like DDM. As addressed in CD detection, the severity can be derived from diverging model performance and dissimilarities between old and new data. The steeper the performance drop, or the bigger the distance between the data old and new data, the higher the severity level of the CD.

The most difficult to answer question is *Where*. Drift regions are regions of conflict between the new and old distribution. Feature vectors located in a drift region are affected by the CD and contribute to the error-rate of the model. Identifying drift regions makes CD more tangible and allows for a more in-depth analysis of model performance. The techniques to make out drift regions are however highly dependent on the employed drift detection model. This means that there is no generalizable way to map out drift regions in the feature space (Lu et al., 2018).

Data understanding doesn’t need to be collected algorithmically alone. Insights into occurring shifts in the data can be visualized as well (Koren, 2009b). Visualizations can be analyzed by data scientists to qualitatively assess CD in a dataset.

CD adaptation. The most conventional way of reacting to CD is to trigger a model retraining process. In the context of MLOps this is referred to as CT. After the defined performance threshold is not met, a new model is trained with new data. There are variations to simple retraining. One is called *ensemble retraining*. After CD is detected, a new model gets trained, which is then added to an ensemble with the old model. Both models are then used to make predictions. Ensemble retraining is suitable for gradual CD, where two concepts are present in the data.

CD adaptation can also be incorporated into the model itself. Gradual forgetting is a technique employed during model training, where training examples are weighted based on how new they are. The result is a model that is more sensitive to the charac-

teristics of newer data (Koychev, 2004). Another way of incorporating CD adaptation is to manually embed temporal effect into the ML model (Lo, Liao, Chang, & Lee, 2018). (Koren, 2009b) identifies temporal dynamics throughout the whole dataset and incorporates them into a CF model. This research was conducted on a Netflix dataset, where it discovered that movies receive better ratings as they age. This observation can be converted into a mathematical formula and be incorporated into the RS model.

CD awareness. In this research, CD awareness is defined as to as the amalgamation of CD detection, CD understanding and CD adaptation.

2.3.6 State of the Art Technology

In order to address the challenges of MLOps, a wide variety of open-source tools and services have been developed in the past years.

TFX. TFX is maintained by Google and used for their own ML projects. TFX compartmentalizes common tasks of a ML lifecycle into separate components. The entire ML process from data ingestion, data evaluation, data transformation, model training, model evaluation, up to model deployment is standardized and modular. Currently there are a total of 10 built-in components that are provided by TFX (Figure 9). The repository of base components can also be expanded by building custom components.

Figure 8: TFX components (Google LLC, 2019a)

These components can be orchestrated and run in a pipeline with support for Airflow and Beam. In addition to the components TFX includes libraries and software components for metadata storage, model creation and model evaluation (Baylor et al., 2017). TFX's central metadata storage is called ML metadata. This SQLite database acts as a model repository and feature store. In addition, it tracks training parameters and the order of component execution to make every pipeline run traceable. ML metadata also saves progress of model training, in order to continue the training at a later time (Crowe, 2019).

Custom components give TFX the flexibility to embed external libraries into the pipeline. However, TFX is clearly optimized to work within the TensorFlow ecosystem. This means that the TF API, TF Model Analysis (TFMA), TensorBoard and TFRS are natively supported by TFX and work without extensive configuration.

The following will provide an overview to the essential components of TFX (Google LLC, 2019b):

ExampleGen is the first component of the TFX pipeline and ingests data in a specified directory. It supports various data types, ranging from TensorFlow's TFRecord to CSV and popular Big Data file formats like Avro or Parquet. The ingested data is then versioned and stored as TFRecord.

StatisticsGen takes the saved data from the ExampleGen component and generates rudimentary statistics from it. This component serves to provide a general overview of the data, which can also be used by consecutive components.

SchemaGen automatically retrieves and infers the schema of the dataset at hand.

ExampleValidator takes the outputs from StatisticsGen and SchemaGen and searches the dataset for anomalies that might impact training performance. This component investigates the data for missing values, data skews between training and serving data, and potential data drift.

Transform is used for data preprocessing tasks. Within this component, arbitrary data transformation operations can be written in a *preprocessing_fn* function. Transform takes in the data output from ExampleGen and the data schema from SchemaGen.

Trainer takes on the model training process of a ML lifecycle. The Trainer component takes in a ML model file with a *run_fn* function, which initializes the training process. For the training and evaluation data either the output from ExampleGen or Transform can be used. Training parameters (e.g., training steps) can be given as arguments. The output of this component is the trained model.

The training process can optionally be augmented with the *Tuner* component, which employs the KerasTuner module to find the best hyperparameters of a model. The output of this component is given to the Trainer component.

Evaluator leverages the TensorFlow Model Analysis (TFMA) library to perform model evaluation. Based on the automated model evaluation, it can be determined whether the trained model yields superior results to the current production model. If the model supersedes the previous model, it receives a "blessing", meaning that it can be deployed to a production environment.

Pusher deploys the model to a specified target (i.e., directory). It takes in the model trained by the Trainer component. If the Evaluator component is used, the Pusher can determine whether to push a model based on the blessing.

Apache Beam is used as the default orchestrator for TFX, which integrates the individual TFX components into a DAG. This DAG can then be triggered to run the entire process without human intervention. Alternatively, TFX can also be run by other workflow management tools, such as Apache Airflow.

KubeFlow. KubeFlow is a solution for large scale ML operations. It provides an infrastructure for a complete MLOps environment in the cloud. It comes with essential software, like notebooks for development, TensorFlow for model creation and Apache Airflow for pipeline orchestration. In addition to built-in software packages, it also provides support for various ML tools, like TFX. KubeFlow is built on Kubernetes which handles

containerization and cluster computing. This cloud infrastructure is highly scalable in nature. This means that model training can be distributed between multiple compute nodes, which accelerates the training process (Kubeflow, n.d.).

AutoML. Automated ML (AutoML) are services that provide end-to-end ML systems. They are provided as software-as-a-service (SaaS) on a cloud platform (e.g., GCP and AWS). The internal workings of AutoML are usually hidden to the user and the system can only be configured through pre-defined interfaces. The goal of AutoML is to provide easy-to-use and intuitive ML solutions, that don’t require extensive programming skills from the user (Karmaker et al., 2022).

3 Goal & Specification

3.1 Artifact

The goal of the DSR methodology is the design and creation of an artifact. The work described in this thesis sets out to create an automated machine learning pipeline for an RS, the main research focus being the implementation of CD awareness into the pipeline to realize continuous training. For evaluation purposes, a specification sheet will be generated on which the artifact will be measured against. The artifact will be compared with the features listed in the specification. Based on the specifications, the final product will be analyzed and the success of this research will be determined. The specifications are structured in *base specifications* and *research specifications*. While both types of requirements are integral to the whole research project, the research requirements have a direct connection to the research question at hand. During the DSR process both requirement types receive equal prioritization as both requirement types make up the whole artifact. A table with all specifications can be seen in *table 1*.

The base requirement consists of creating an SotA RS inside a minimal training pipeline. In addition, a suitable dataset for this research is selected. Throughout the iterative development process this pipeline will be expanded in order to fulfill the research objective of this work.

Table 1: Artifact specification table

Specification type	**Specification**	**Description**
Base specification	Recommender system	The artifact has an SotA RS and a suitable dataset to run on.
	Base pipeline	The artifact has a pipeline to run a basic ML lifecycle on. This base pipeline serves as a baseline and will be iterated over.
Research specification	CD detection	The artifact can detect CD.
	CD understanding	The artifact can extract information out of the CD.
	CD adaptation	The artifact can react to CD.
	Integrated pipeline	The final artifact is one pipeline that can be executed in one go
	Automated pipeline	The pipeline can run without hu-

		man intervention

There are three specifications related to CD for this artifact, which is derived from the 2018 paper “Learning under Concept Drift: A Review” (Lu et al.): CD detection, CD understanding and CD adaptation.

Each of these components amount to what is define in this paper as CD awareness. All three specifications will be explained in further detail in the literature review.

Besides the concept drift specifications, there are also pipeline conditions in the research specifications that envision the artifact to be one unified and automated process.

These specifications serve as an aide during the design and development phase and will be later used to qualitatively evaluate this PoC.

3.2 Procedure

Throughout this project various tools are used to track, organize and document this work. As part of this research, a GitHub repository was created, which contains both the thesis Word document and the software artifact.[9] GitHub and Git are used for version control and enable work on different systems. It also serves to make the development of this research transparent and traceable. Internally, a GitHub Project Kanban board is used for project management. There, the project is broken down into individual tasks, all of which have their progress tracked. This way it is possible to gain an overview of the current status of this project and plan future development steps.

The overarching process of the artifact creation is structured in underlying phases, as seen in Figure 10. Each phase will be developed in a separate branch and then merged into the main branch, as is convention in software development.

The initial phase of this project is dedicated to the set-up of the artifact environment. This phase encompasses every necessary step to build a software environment on which the artifact will run on. The individual tasks would consist of choosing the python version, setting up a work directory and installing needed packages and other software.

After setting up the environment, an appropriate public dataset is selected for this project. If needed, a small data preparation task will be done to make the data ingestible by the MLOps tools used in this project.

Next, a base MLOps pipeline with a placeholder RS is built. It should fulfill the basic functionality of an MLOps pipeline and serves as the foundation for further development steps.

[9] https://github.com/MyPetOctocat/bachelor_2022

The following phase involves swapping the placeholder RS for one SotA RS discussed in chapter 2.2.

After implementing the new RS, the base pipeline then gets incrementally expanded by additional components. Each pipeline component that gets added will be tested for compatibility. Depending on what components work and don't work, design changes to the artifact need to be made.

Having implemented the RS and the pipeline, an intermediate evaluation of the current pipeline will be done. The objective in this phase is to find out the capabilities of this pipeline in relation to the research question. What components can be leveraged to fulfill the objective of this work? Accomplishing these phases should fulfill the base specifications detailed in the previous chapter.

The following phase consists of incorporating CD awareness into the pipeline. With the knowledge base from scientific papers and information about the environment from the whitepapers, a solution to CD will be engineered. This process will run iteratively and in accordance with the DSR methodology. Each iteration of this phase concludes with an evaluation, which will determine whether another iteration is initiated.

Once the artifact reaches a state where no additional iterations are done, a final evaluation of the artifact will be made, which concludes this research project.

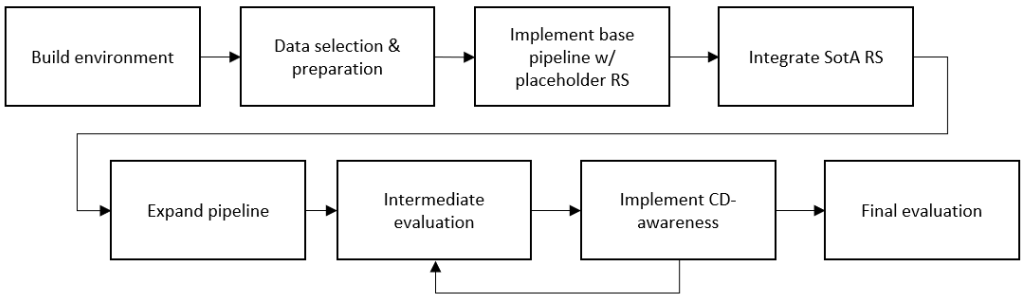


Figure 9: procedure of artifact development

4 Design & Development

4.1 Environment

4.1.1 Hardware & Software environment

This research project is conducted on a deep learning compute cluster provided by the Media University Stuttgart (Theodoridis & Grießhaber, n.d.). The hardware resources are allocated by Slurm, a cluster management software. This way, multiple users can access powerful hardware to train large DL models. The DL cluster consists of multiple individual servers, which can be classified as either a CPU partition or GPU partition. While CPU compute nodes are sufficient for development and data preparation tasks, GPU acceleration can be leverages to rapidly compute tensor calculations.

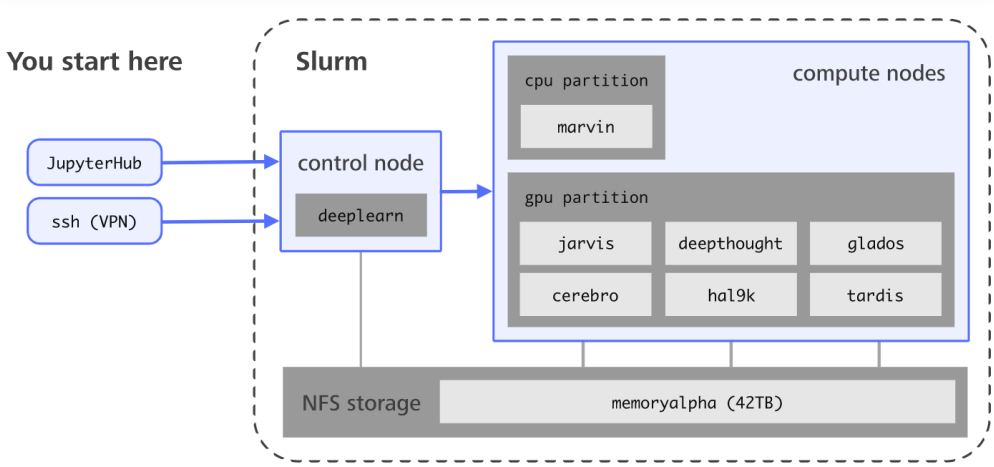


Figure 10: Infrastructure of the HdM deep learning cluster (Theodoridis & Grießhaber, n.d.)

These servers are accessed via the control node, which acts as the entry point to the cluster. The control node can either be addressed via a web browser through JupyterHub or with secure shell (SSH), in combination with a VPN. The cluster has a shared network file system (NFS), where the files from all users are stored (Figure 11). In order to start a server, the user needs to select either a CPU or GPU instance. Depending on the selection, the user gets assigned a corresponding cluster node. Each time a server instance is started, an Ubuntu image gets mounted to the users work directory. This image comes with Anaconda preinstalled, in order to create and manage Python environments.

Python will be used as the main programming language of this project, as it is built-in into Anaconda and supports a wide array of data science packages (e.g., TensorFlow, NumPy, pandas) to conduct this research. Specifically, Python version 3.8 is chosen.

During the assessment of a suitable research environment, KubeFlow was briefly considered. As mentioned in the State of the Art Technology chapter of “MLOps”, KubeFlow would provide an all-in-one environment for this research. KubeFlow is a viable solution for large scale MLOps in an enterprise setting but wouldn’t be appropriate for the PoC that is developed in this research. The set up and configuration of the environment would exceed the scope of this project. KubeFlow instantiates various containers with ML software, which are intelligently distributed across a server cluster with Kubernetes. While this makes for a scalable productive system, it also has high hardware requirements.

For this artifact, a lean software stack of few selected tools is sufficient to conduct this research.

4.1.2 Tools & Frameworks

Most MLOps systems are an amalgamation of different tools. MLOps pipelines consist of individual components (e.g., model training, deployment, monitoring), which use specific software libraries and frameworks to perform their task. A challenge in building pipelines is the integration of these components into one coherent end-to-end pipeline. Therefore, it is not only important to select tools based on their own merits, but also evaluate their compatibility with other tools used in the pipeline. For this research project, usability, familiarity, and compatibility are key factors in determining what tools should be used for this artifact. Throughout this work, multiple libraries need to be applied in order to realize an end-to-end pipeline. Due to the time constraints of a bachelor thesis and the diversity of an MLOps system software stack, it is particularly important to select packages that are intuitive to use, learn and integrate into the artifact.

The artifact of this work can be structured into three main components: The RS, the CD awareness, and the pipeline. Each component requires its own set of tools which are specific to their task. In the following, the tools used for each component are established, which makes up the software stack of this research project.

Recommender System. For this project TensorFlow Recommenders is used. TFRS is built on Keras, which provides easy-to-use APIs to construct DL models. A wide array of built-in methods makes Keras a flexible and powerful library. Detailed information about the model architecture can be retrieved, which makes debugging and model-understanding easier. In addition to the standard Keras functionality, TFRS also has RS specific features to create comprehensive SotA recommendation systems. This helps streamline the RS creation process, as common operations for RSs are provided as methods. The RS implemented in this research is a DCN, which is a SotA architecture proposal from Google and makes use of cross layers to extract and learn explicit cross-feature interactions in the dataset (Wang et al., 2021). TFRS comes built-in with cross layers, which behave similarly to other Keras layers. This makes them applicable to any Keras model architecture and enables one to create SotA models, with the intuitiveness that the Keras API provides.

CD awareness. For CD awareness, a custom solution is written. This component of the MLOps system therefore doesn't utilize any dedicated concept drift detection library. In order to analyze and evaluate monitoring data, *pandas* is used. Pandas is a python library for data processing and data analysis (pandas, n.d.). This library loads data (e.g., CSV files) into *Dataframes*, where it can be analyzed and manipulated with an extensive selection of methods. The flexibility of Dataframes makes data analysis intuitive and powerful.

For large-scale data analysis and processing operations, *pandas'* performance becomes a bottleneck. Since this artifact is a PoC and won't be working with production-scale datasets, *pandas* for data processing is sufficient. The goal of this project is to design and outline a solution for how CD can be effectively detected.

Pipeline. The orchestration and pipeline management of the artifact is realized with two core components: TFX and Apache Airflow. TFX is used in this project to carry out the entire model training process, ranging from ingestion of training data, over model training, to model deployment. TFX compartmentalizes essential tasks of a ML lifecycle into separate components. The advantage is, that the built-in components from TFX are compatible among each other. TFX manages the interactions between each component leaving the user to configure each component to their specifications. This makes development of the ML pipeline more manageable, as components can be individually tuned, without interfering with the rest of the pipeline. TFX automates various tasks, like data ingestion, computation of statistics and anomaly detection of training data, without the need of additional configuration by the user. A lot of these default settings provided by TFX are sufficient for this PoC artifact and provide a baseline for future iterations. Since TFX is developed by Google and the TensorFlow team, it shares compatibility with the TensorFlow ecosystem, including TFRS. It can interpret TF and Keras models and run evaluation steps on them. TFX also has an extensive repository of public documentation available, among these are a lot of practical user-guides and tutorials. This information is useful in designing the artifact and helps overcome challenges that are faced using this framework.

The underlying orchestration of the TFX components is managed by Apache Airflow in this research. The default orchestrator for TFX is Apache Beam but has been changed to Airflow. The reason being, that besides the TFX component, the artifact has a second pipeline in the production environment, where the CD detection is orchestrated. In order to unify both processes, the same workflow management tool is used. The goal of this project is an integrated and automated pipeline and by using Airflow for both pipelines, they can be combined into one complete pipeline. Airflow is a suitable tool for this research, as it comes with a Web-UI, which provides an overview of the pipelines. From this web interface, pipelines can be manually triggered, their runtime overseen, and errors traced back through its log-repository.

4.1.3 Working Directory

The working directory of the artifact is located in `bachelor_2022/artifact` (Figure 13). Within the working directory are other subdirectories. The green folders seen in Figure 13 are not tracked as they do not contain any source code.

`airflow_pipelines` stores all pipelines executed by Airflow.

`data` is a subdirectory where all the datasets before a pipeline are stored.

`data_fetch` contains scripts for data retrieval and data preparation. These scripts can be executed to download files and prepare them to make them readable by the employed software tools. The contents inside `data` are the product of these scripts.

`pipeline_scripts` contain all files related to pipeline execution and pipeline orchestration.

The source code for the ML models is saved in `model_source`. The pipelines use these files to train the ML model.

`pipeline` is the directory where all training pipeline outputs are then stored.

In the `production` folder the productive environment is simulated. Inside this directory resides the prediction service, monitoring and CD awareness of this project.

`sandbox` serves as an environment where concept and test code are saved. These scripts aren't part of the artifact and thus are located in a separate folder.

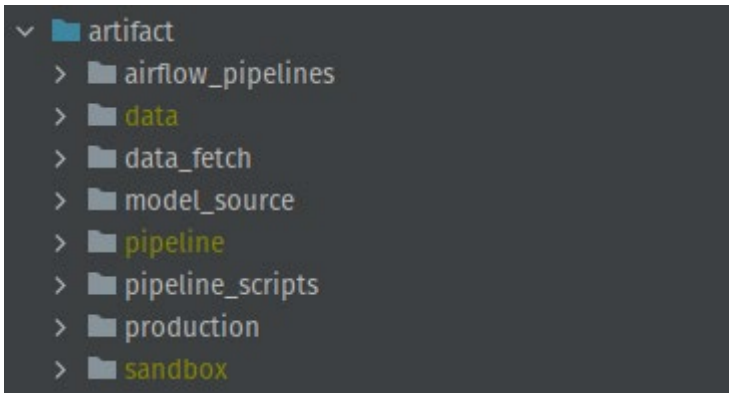


Figure 11: Working directory and its subdirectories

4.2 Data

4.2.1 Dataset Selection

This research is not conducted within a company environment, which means that a public dataset needs to be chosen in order to realize the artifact. There are various public datasets available that are used for benchmarking by the scientific community. One of the most popular datasets for RSs are the MovieLens datasets from GroupLens (Harper & Konstan, 2016). GroupLens collects movie ratings of users since July 1998,

that are then continuously released to the public. They offer datasets of varying sizes ranging from 100.000 up to 25 million ratings, wherein each user has at least 20 movie ratings. An overview of the different datasets can be seen in Table 2.

Typical for a recommender dataset, MovieLens contains information about user and item interactions. The two smallest datasets, 100k and 1m, have additional demographic information about their users. These dense features make them a viable choice considering SotA RSs that can leverage both sparse and dense features for predictions.

This project sets out to create a PoC MLOps pipeline. This means that RS benchmarking isn't part of this research. For this reason, larger datasets, like 20m and 25m, have not been considered, as the training process would have taken up a significant amount of time. Due to the time constraints of this project, it is important to be able to make rapid iterations to the pipeline. For this reason, the MovieLens 100k dataset is considered sufficient for this thesis.

Table 2: Overview of MovieLens datasets (GroupLens, n.d.)

Name	**100k**	**1m**	**latest-small**	**20m**	**25m**
Ratings	100.000	1.000.209	100.836	20.000.263	25.000.095
Users/ Movies	943/ 1.682	6.040/ 3.952	610/ 9.742	138.493/ 27.278	162.541/ 62.423
Dense user features	Yes	Yes	No	No	No
Timespan	1997-1998	2000-2003	1996-2018	1995-2015	1995-2019

4.2.2 Dataset Description

The MovieLens 100k dataset can be divided into 3 parts. The first part maps the user-item interaction. It consists of a user id, movie id and a resulting rating between 1 and 5, as well as a Unix timestamp of when the user-rating was logged. These represent the sparse features of the dataset. The other parts hold dense features about the user and the movies. The demographic data entails information about the user age, their gender, their occupation and the zip code. The movie data is made up of the movie name, its release date, a link to the IMDb[10] page of the movie, and columns for each

[10] IMDb (Internet Movie Database) is a database of entertainment media. It provides information about movies, TV shows and videogames. (IMDb (2022)

genre. The genres are encoded in a binary fashion, meaning that genres belonging to the movie are assigned the value 1, otherwise it's set to 0.

4.2.3 Data Preparation

For the data preparation task, `movielens_csv_generator.ipynb` is used. This script is run once to download the MovieLens dataset and preprocess it in order to be used by the RS. In the following, this script is explained in more detail.

First, the MovieLens 100k dataset is downloaded from the GroupLens file repository (GroupLens, n.d.). The dataset comes in a zip-file, which needs to be extracted. After the contents have been unpacked, the zip file is deleted. This process is done with the following inline commands in the Jupyter notebook:

```
# Fetch data (only run for the first time)
!wget https://files.grouplens.org/datasets/movielens/{ds_name}.zip
!unzip {ds_name}.zip
!rm {ds_name}.zip
```

The extracted files are saved in the `downloads` folder. The dataset is divided into separate files. In order to create a complete dataset, the files need to be concatenated. For this research, the user demographics will be joined together with the sparse data of the user-item interaction. The individual parts of the dataset are loaded into pandas and merged into one Dataframe. Dataframes are objects that hold data in form of a table. Dataframes support a wide array of operations to transform and manipulate the data, which is the reason it is used for this data preparation task.

```
# Augment data_df with user_df
df = data_df.merge(user_df, on='user_id')
```

	user_id	movie_id	user_rating	timestamp	raw_user_age	user_gender	user_occupation_text	user_zip_code
0	196	242	3	881250949	49	M	writer	55105
1	305	242	5	886307828	23	M	programmer	94086
2	6	242	4	883268170	42	M	executive	98101
3	234	242	4	891033261	60	M	retired	94702
4	63	242	3	875747190	31	M	marketing	75240

Figure 12: Sparse rating features and dense user features merged

The dataset used by the RS will use 5 features, with the movie rating as the label:

- *user_id* (sparse)
- *movie_id* (sparse)
- *raw_user_age* (dense)
- *user_gender* (dense)
- *user_occupation* (dense)

- *user_rating* (sparse, label)

Since SotA RSs can leverage both sparse and dense features to train a model, it was a priority to create a dataset that incorporates a mixture of both types of features. In order to simplify the data ingestion process for the RS, all features should be categorial and ordinal values. All features are already categorical besides `raw_user_age`, which is continuous. In order to make `raw_user_age` categorical, each user age is assigned to a specific age group, seen in Table 3. These age cohorts are derived from other MovieLens datasets (GroupLens, n.d.).

Table 3: Age cohorts

Age group	1	18	25	35	45	50	56
Age range	<18	18-24	25-34	35-44	45-49	50-55	55<

In the next step all features are integer encoded. This is done to easily generate a category vocabulary for the RS, which is explained in the following chapter. In Figure 16 the final dataset can be seen.

	user_id	movie_id	user_rating	user_occupation	user_age_cohort	user_gender
0	196	242	3	1	1	1
1	305	242	5	2	2	1
2	6	242	4	3	3	1
3	234	242	4	4	4	1
4	63	242	3	5	5	1
...	...	...	...	...	...	...

Figure 13: Pipeline dataset

After data preparation is done, the Dataframe is saved in CSV format inside the `recommender-systems` folder. Throughout this project different data preparation approaches and data formats have been explored.[11] Besides CSV, TensorFlow's TFRecord format has been considered as well. Compared to CSV, it is more scalable and supports parallel-training in conjunction with TensorFlow. TFRecord is however only natively supported within the TensorFlow ecosystem. Ultimately, CSV has been chosen as the data format, because of its general compatibility with different tools and its ease of use with libraries like *pandas*. Since the relatively small MovieLens 100k dataset is used for this work, performance isn't a detrimental factor in the selection of the file format.

[11] The different data preparation approaches can be reviewed in the repository of this research under `artifact/data_fetch`

4.3 Recommender System

4.3.1 Design

As stated in chapter 4.1.2, a DCN is developed as the RS for this artifact. Inside the working directory, the model source code can be found in the `model-source` folder. The file is named `dcn_ranking_training.py`. The base structure of this model file is derived from the official TFX-TFRS documentation, which is made up of two classes (Google LLC, 2022): `RankingModel` and `MovieLens`. `RankingModel` contains both, the data embedding task and the model architecture. `MovieLens` encompasses the loss and evaluation metrics for the training process. The RS is then instantiated, trained and saved into a specified directory with the `run_fn` function. `run_fn` also contains all parameters that are used for model training. Compared to the baseline model file from the Google documentation, this implementation changes the model architecture from a normal NN to a DCN. This model is also further expanded to train with dense features, like user demographics, in addition to the sparse rating features. Finally, `run_fn` is expanded to also extract and save additional model information. This is done to make the trained RSs more explainable.

4.3.2 RankingModel

`RankingModel` gets instantiated inside the `MovieLens` and passes training data to the model through the `call` method. The training data is first fed through the embedding layers and then through the DCN. The `call` method returns the result of the output layer back to the instantiation of `MovieLens`.

Feature Embedding. The first task of `RankingModel` is dedicated to feature embedding. The embedding process maps categorical input values of a feature into a dense vector of fixed size. Contrary to sparse feature embedding, like one-hot encoding, the embedding layers in NNs are weights that are learned during the training process. The embedding layer is initialized with random values, which are adjusted through back-propagation. After training, the feature embeddings carry semantics in relation to the RS task, which is represented by the values of the weights within the vector space (López-Sánchez, Herrero, Arrieta, & Corchado, 2018).

The advantages of embedding layers over one-hot encoding[12], is that the vectors are dense. One-hot encoding is highly inefficient with features that have a lot of unique categorical feature values, like user and items of an RS dataset. Also, relationships and similarities between different feature values can be represented in the vector space.

[12] One-hot encoding arranges categorical features into a vector of categorical columns, which are binary encoded. The value of 1 is given to the column of the present category, all remaining columns are assigned the value 0.

The basic operation of feature embedding used in this project is outlined in Figure 17. The categorical values (e.g., string value or ID value) are indexed in a vocabulary. The vocabulary serves as a look-up table, which is used to access the right feature embeddings for the corresponding input value. The embedding layer is a $s \times d$ matrix, where s represents the vocabulary size of the feature and d refers to the chosen dimensionality of the embedding. In order to retrieve the right embedding from the layer, the matrix is multiplied with a one-hot encoded vector of the input value. The product is a vector of size d, which contains the embedding of the input value.

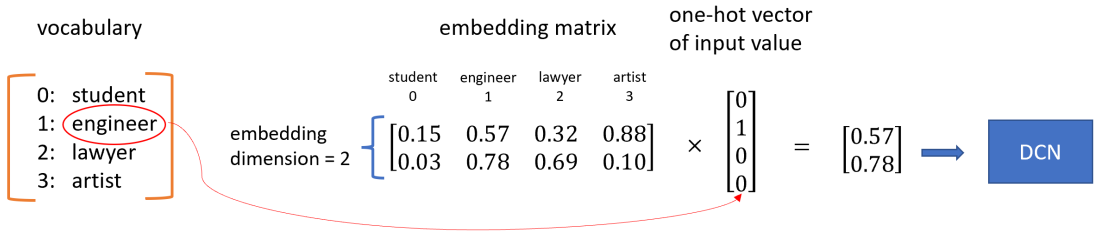


Figure 14: Simple embedding example for feature “occupation”

All features in this dataset are treated as categorical variables. This means that for every feature, an embedding layer is created. This paragraph will describe the implementation of the feature embedding inside the `RankingModel` class.

At the beginning of the class, variables are defined that are going to be used by the embedding layers. First, the dimensionality of the feature embedding is declared in variable `embedding_dimension`. For this work, the embedding size of the baseline model is used, which is 32 dimensions. This embedding dimension will be applied to every feature in the dataset. Next, a vocabulary for each feature is declared. The vocabulary is represented in a NumPy array of the length of unique feature values. By filling the array with incrementing values, the index is created. We recall that all features of the dataset are integer encoded, as described in chapter 4.2.3. For this reason, the vocabulary doesn’t need to be a dictionary, as the values in the array already correspond with the encoded values from the dataset (Figure 18).

```
33      # Define the dimension the feature values should be embedded in
34      embedding_dimension = 32
35      self.embedding_dims = embedding_dimension
36      # Create np array with incrementing values as the vocabulary
37      unique_user_ids = np.array(range(943)).astype(str)
38      unique_movie_ids = np.array(range(1682)).astype(str)
39      unique_occupation_ids = np.array(range(21)).astype(str)
40      unique_gender_ids = np.array(range(2)).astype(str)
41      unique_age_ids = np.array(range(7)).astype(str)
```

Figure 15: Declaration of embedding dimension and vocabularies inside RankingModel

Now, that the embedding dimension and vocabulary have been defined, the embedding layers are configured. Since the features are integer encoded, all embedding layers receive an integer value as an input, which is defined in the Keras `Input` layer. The input is then converted into a string value, which is passed to the `StringLookup` Keras layer. This function searches for the feature value inside the vocabulary and returns its index. As seen in Figure 17, the index is used to create a one-hot vector of the input value, which is then passed to the `Embedding` layer. The embedding layer is multiplied by the one-hot vector from the previous layer, which outputs the feature representation of the input value. The embedding is then ingested by the DCN (Figure 19).

```
44      ## String values embeddings
45      # Compute embeddings for users.
46      self.user_embeddings = tf.keras.Sequential([
47          tf.keras.layers.Input(shape=(1,), name='user_id', dtype=tf.int64),
48          tf.keras.layers.Lambda(lambda x: tf.as_string(x)),
49          tf.keras.layers.StringLookup(
50              vocabulary=unique_user_ids, mask_token=None),
51          # Create embedding layer of dimension 943x32
52          tf.keras.layers.Embedding(
53              len(unique_user_ids) + 1, embedding_dimension)
54      ])
```

Figure 16: Implementation of the embedding layer

DCN. The implementation of the DCN in this research uses a total of 4 layers. The model in its entirety can be seen in Figure 20. This model serves as a PoC for the cross layer, which is part of the TFRS library. In order to ensure reproducibility of the model, all layers within the DCN are initialized with a seed. This means that the weights before model training are the same throughout all training runs. As proposed by Wang et al. (2021), the cross network is positioned before the NN and after the embedding layer. It therefore receives the feature embedding as its inputs. As mentioned in chapter Deep & Cross Networks, the degree of cross feature interactions explicitly learned, can be controlled by the number of layers in the cross network. This model architecture uses one cross layer, which declares that the learned feature interactions to a degree of 1. This means, that the CN considers interactions between one feature to another one. The cross network is then followed by two dense layers and an output layer. For both dense layers, the ReLU activation layer is used. The first NN layer consists of 256 neurons, while the second layer has a total of 64 units. The output layer returns a float value, which represents the predicted rating of the user for a movie.

```
95      # Cross Layer
96      self.cross_layer = tfrs.layers.dcn.Cross(kernel_initializer=tf.keras.initializers.RandomNormal(seed=1)) # Use seeds
97
98      # Compute predictions.
99      self.ratings = tf.keras.Sequential([
100         self.cross_layer,
101         tf.keras.layers.Dense(256, activation='relu', kernel_initializer=tf.keras.initializers.RandomNormal(seed=1)),
102         tf.keras.layers.Dense(64, activation='relu', kernel_initializer=tf.keras.initializers.RandomNormal(seed=1)),
103         tf.keras.layers.Dense(1, kernel_initializer=tf.keras.initializers.RandomNormal(seed=1))
104     ])
```

Figure 17: Implementation of the DCN

4.3.3 MovieLens

`MovieLens` inherits from the TFRS Model class, which is a wrapper for the Keras Model class. This base class lets the developer define custom training and test losses, by implementing three methods in the child class:

In the `__init__` method, a Keras model and the loss metrics are declared. `self.ranking_model` is an instantiation of the RankingModel class, discussed in the previous chapter. `self.task` is an object of the Ranking class from the TFRS module. This class can be given a loss function as a parameter with which the training loss is calculated. In addition to the loss function, additional evaluation metrics can be given as parameters. This artifact uses mean squared error as the loss function and root mean squared error (RMSE) as an evaluation metric, which can be seen during the training process (Figure 21).

```
60/60 [==============================] - 7s 110ms/step - root_mean_squared_error: 0.3033
s: 0.0921 - val_root_mean_squared_error: 1.2647 - val_loss: 1.6919 - val_regularization_
```

Figure 18: Metrics during model training

The second method in MovieLens is `call`. This method takes in the features of the training data and gives it to the RankingModel in order to compute the predictions.

These predicted labels are then given to `compute_loss`, where they are compared against the true labels. These two values are passed to the task object, which executes the call method in `Ranking` and returns the loss value and metrics.

4.3.4 Post-Training Actions

Metadata collection is a key task in MLOps systems. It can give ML engineers an overview of trained models, help identify weaknesses in a model and improve model understanding. In order to satisfy the need for model metadata in an MLOps environment, this research project retrieves and saves custom information after every training. These are referred to in this paper as post-training actions (PTA). Three PTAs are implemented in this artifact.

Visualization of model architecture. The first PTA is a visualization of the model architecture. Keras provides a method to visualize its sequential models, which reads the

layers of the model and maps them to a plot. The model architecture in this project is defined inside the RankingModel class, which is instantiated in self.ranking_model of the MovieLens class. The ratings variable is passed to the Keras plot_model method to generate the visual representation of the model. By giving the plot the same name as the model, it becomes clear to which model the visualization belongs to. In Figure 22 an output of this PTA can be seen.

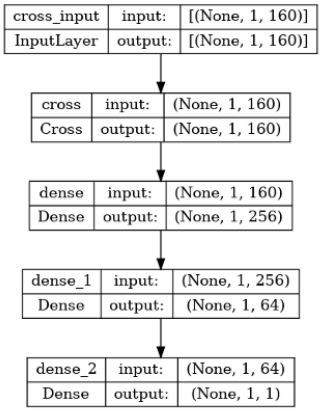


Figure 19: Model architecture visualization

The intention behind this PTA, is to provide a model overview to ML engineers. Models can become complex and it is therefore beneficial to have visual information about their composition.

Visualization of model training. The second PTA has the objective to give insights into the training process of a model. This is done with TensorBoard. TensorBoard reads log files generated during training and visualizes them in a dashboard. Conventionally, TF logfiles are stored in a temporary folder. By configuring a custom path, these training logs can be saved permanently. These files can then be visualized by launching TensorBoard in the configured logfile directory, which opens a web interface. There, the training course of all models can be reviewed and compared. In Figure 23, the training progress of a model is seen through the dashboard.

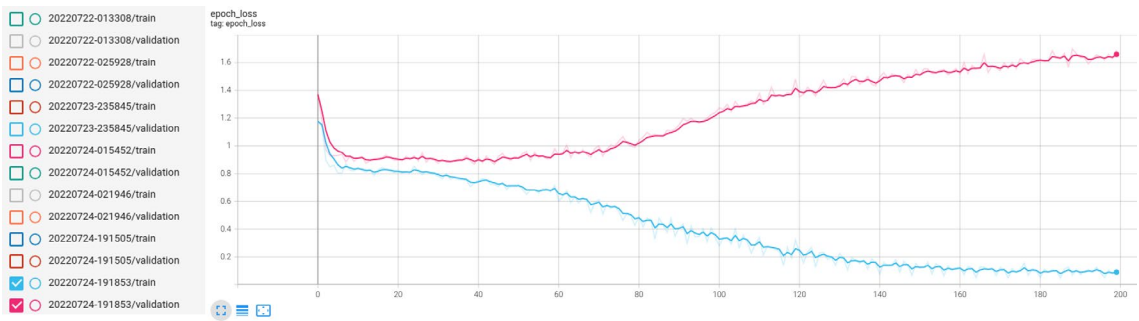


Figure 20: TensorBoard training and evaluation visualization

Visualization of learned cross-features. Model understanding is a non-trivial task for large neural networks. Because of the number of parameters, models become hard to explain. Fortunately, cross networks are much easier to interpret, as their weights are always allocated to specific cross interactions. The interpretability of these weights can be leveraged to improve model understanding. As part of the second PTA, the cross network is visualized to present information about the feature interactions that the model has learned. The final plot is a $m \times m$ matrix where m is the number of features of this model. Each matrix holds an element, which represents how strong the learned cross interactions between two features are (Figure 24). This plot is created with seaborn and saved with the name of the trained model to make it retraceable.

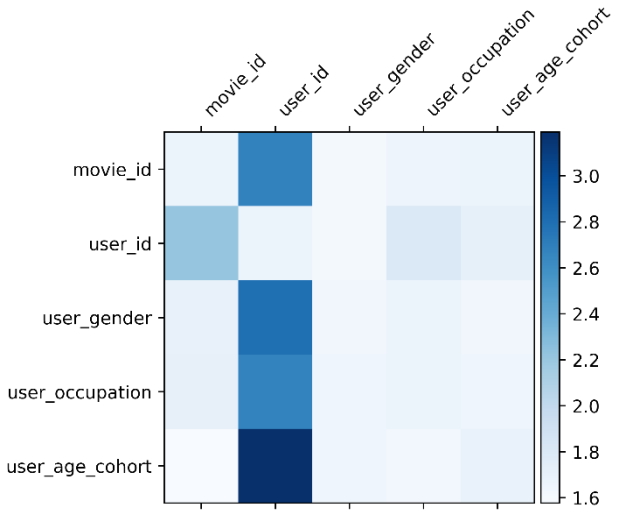


Figure 21: Plot of the learned feature interactions in a cross layer

In the following, it is explained how the cross layer is translated into a visualization. In Figure 25 the console output of a trained cross layer is shown. The cross layer of this network is a 160×160 matrix. 160 corresponds to the output dimension of the previous layer, which is the embedding layer. In the embedding layer, each of the 5 features are translated into a 32-dimensional vector representation. Multiplying the number of features with the 32 dimensions results in an output of 160 dimensions. In order to map cross interactions between features, a square matrix of the output dimensionality is created. In this case, each 32×32 slice of the cross layer represents a specific cross interaction between two features. In order to bring the 160×160 matrix into a 5×5 matrix, the weights representing one cross-feature interaction need to be summarized into a single value. This is done by calculating the norm of each 32×32 sub-matrix of the cross layer. In this case the Frobenius norm is used, which calculates the root sum of squares of the matrix.

```
<tf.Variable 'cross_2/dense/kernel:0' shape=(160, 160) dtype=float32, numpy=
array([[ 0.00856209, -0.00121698, -0.04012398, ..., -0.02011229,
         0.02701266,  0.02212355],
       [ 0.05741805,  0.03309018, -0.0272197 , ..., -0.0102978 ,
        -0.01930288,  0.00873917],
       [-0.04755176, -0.01540281, -0.00824679, ...,  0.03096272,
        -0.03412312,  0.04841772],
       ...,
       [-0.01216198, -0.06522709,  0.07514991, ..., -0.01814271,
         0.05718886, -0.00441373],
```

Figure 22: Console output of cross layer after training

4.4 Concept Drift Awareness

4.4.1 Design

The CD awareness process is structured in two parts. The first component is the prediction service and the second is monitoring. The prediction service simulates a ML application that predicts ratings for users, by using the production model. These predicted ratings, as well as the actual user ratings are then stored to be used by the monitoring component. Monitoring entails the CD awareness of the artifact, which consists of CD detection, CD understanding and CD adaptation, as mentioned in chapter Concept Drift. For drift detection an error rate-based approach is taken. This means that the model performance will be the determinant for CD detection.

4.4.2 Prediction Service

The prediction service is initiated through `prediction_service.py` in the `production/prediction_service` directory. There the production data is loaded into a Dataframe. The production data contains the same features as the training data. Two additional columns are added to the Dataframe: `pred_rating` and `pred_int_rating`. Both columns are filled with zeroes as placeholder values. `pred_rating` is later filled with the raw output values by the model. `pred_int_rating` is going to contain the model output in form of the rating system of the dataset. In the case of the MovieLens dataset it is an integer between 1 and 5. This Dataframe is passed as an argument to the `batch_predict` function in `batch_predictor.py` file. This function iterates over the dataset and calls `prediction_server` in `predictor.py`. This function takes in the RS model and a single user-movie entry as arguments and returns the output of the model. This rating then replaces the filler value in the `pred_rating` column. For `pred_int_rating`, the model output is converted into the 5-star rating system, by rounding the float value to an integer of an interval of [1;5]. The `batch_predict` function returns the complete dataset with its predicted ratings. In the last step, the Dataframe is saved to be used for the CD detection task in `monitoring` (Figure 26).

movie_id	user_rating	timestamp	user_occupation	user_age_cohort	user_gender	pred_rating	pred_int_rating
52	3	876176979	1	3	0	2.589300	3
849	4	879848590	10	2	1	3.834248	4
79	4	879438984	9	5	1	4.254503	4
139	2	884134134	9	2	1	2.310845	2
444	3	884134075	9	2	1	3.037222	3
762	5	882141336	7	1	1	2.690102	3
62	2	880473583	9	3	1	2.305300	2
275	4	883530521	10	5	1	3.600219	4
165	5	885598961	2	3	1	4.934212	5
237	3	879438911	7	6	1	3.005077	3
1110	3	880037334	14	1	0	5.254304	5

Figure 23: Output of the prediction service

4.4.3 Monitoring

In order to meet the requirements of this research project, CD detection, CD understanding and CD adaptation need to be integrated into the monitoring process. Instead of using a third-party drift detection library, a custom solution is developed, which shares similarities with algorithms, like DDM. The monitoring is done on the data provided by the prediction service. The model performance development is derived from the actual user ratings and the predicted ratings of the model. For this, RMSE is chosen as the evaluation metric for CD.

The source code for the CD evaluation task is located under the `airflow_pipelines` directory. The monitoring script that holds the CD awareness functionality is `cd_awareness_pipeline.py`. This script is located inside the Airflow pipeline folder because it is integrated into a pipeline, which will be covered in the next chapter. Like in the prediction service, the data is loaded into a pandas Dataframe, in order to conduct the necessary data processing steps. First, a column is added to the Dataframe, where the Unix timestamps are converted into date objects. With a date column, data can be easily grouped by a desired time unit (e.g., month, or year), which is needed to calculate the RMSE in the next step. The calculation of RMSE is done with the `rmse_calc` function in `rmse_calculator.py`. This function takes in a Dataframe with three columns: A date column, a column with the user rating and another column with the predicted ratings. Another argument is passed to specify the time aggregation. For this project, the data is grouped by the year. This function then returns a pandas Series with the calculated RMSE values for every year. This function is applied for both the float prediction and the integer prediction, which are then merged into one Dataframe (Figure 27).

date	rmse_float	rmse_int
1997-12-31	0.838945	0.777029
1998-12-31	0.811808	0.785507

Figure 24: RMSE values of the production data grouped by year (rmse_df)

This Dataframe is used as an input to function `cd_detector`, which runs the CD detection task. Furthermore, two threshold parameters are defined. Should these thresholds be surpassed, it is assumed that CD has occurred in the data (Figure 28). `delta_threshold` defines the allowed RMSE increase from one time window to the next time window (i.e., from one year to the next year). `absolute_threshold` denotes the maximum absolute RMSE value a window is allowed to have. This threshold is a safeguard for incremental and gradual CD, where shift in data is slow and might not be picked up by the delta threshold. This function returns a tuple with a Boolean and a list. If the Boolean is set to False, CD has been detected. The list contains two elements, which holds the dates of the window, CD was detected in. The first element holds a date for the detected delta CD, the second element holds the date for the detected absolute CD. This way it is possible to also see when in the dataset CD has occurred.

```
24  # CD detection (determine whether CD occurred in the data)
25  no_cd = cd_detector(rmse_df, delta_threshold=0.1, absolute_threshold=1.5)
```

Figure 25: cd_detector

After the CD detection is run, the results are saved in form of a text file in the `evaluation_outputs` folder. If the filename is “OK.txt”, no CD has been detected. If CD has been detected, a file named “CD_DETECTED.txt” is created. Both text files hold a timestamp of the CD detection run. “CD_DETECTED.txt” holds additional timestamps of the CD occurrence in the dataset (Figure 29).

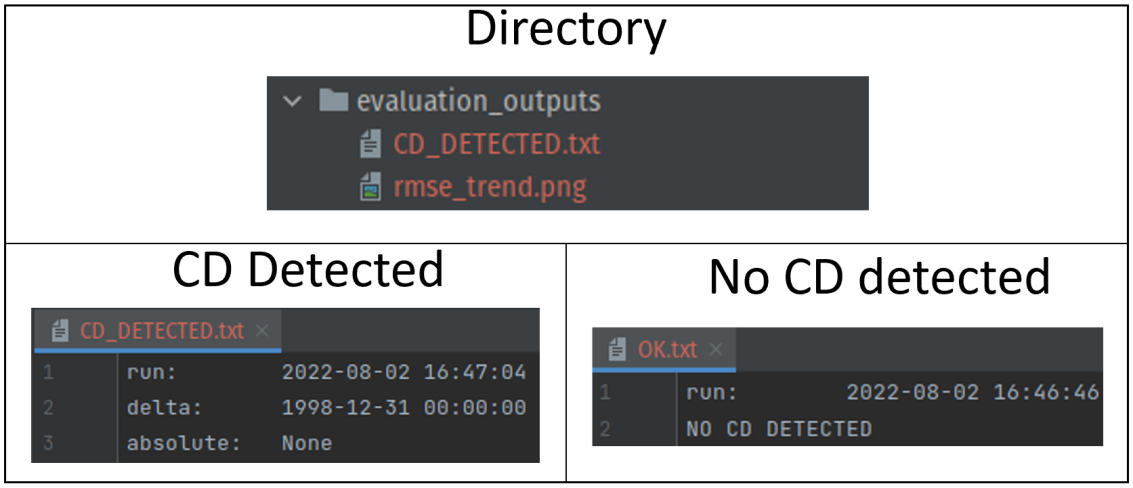


Figure 26: CD detection result

After the CD detection task, a graph containing the progression of the RMSE is created. In order to have more granularity in the visualization, another RMSE table is created, this time with a monthly aggregation. The Dataframe is then visualized with seaborn and saved in `evaluation_outputs` (Figure 30). This visualization is the implementation of CD understanding in this artifact. Through this graph, the trends of the CD can be analyzed and retraced.

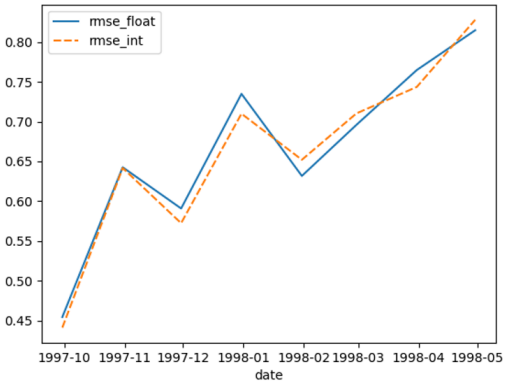


Figure 27: Graph of CD trend

CD adaptation is realized through orchestration with Apache Airflow and is explained in greater detail in the next chapter. The CD adaptation in this artifact is implemented through model retraining-trigger. If the `cd_detector` function returns `False` (i.e., CD has been detected), a model retraining process is initiated.

4.5 Pipeline

4.5.1 Design

This research project encompasses two processes in total: The model *training process* and the *CD evaluation*[13] process. For this work, both operations are automated into individual pipelines and subsequently joined together to form an integrated continuous training pipeline. The training process is realized with the TFX framework. With TFX, Google already worked out the essential tasks for a training pipeline and integrated them into individual components. These provided TFX components serve as a guideline during the implementation and are later used for the artifact evaluation. The baseline TFX pipeline consists of data ingestion, model training and model deployment. This marks the starting point of the TFX pipeline and from there other components are incrementally added. The CD evaluation process is orchestrated with Airflow, which is triggered in a repeated time interval.

It is a conscious design decision to first map both processes into two different pipelines, instead of building one large pipeline from the start. This is done to meet the software engineering principle of *separation of concerns*. Like in conventional software development, modularization makes the development of ML systems more sustainable and scalable. By separating both processes, development on a pipeline can be done without interfering with the other pipeline.

Both standalone pipelines are then linked by a third, overarching pipeline. This pipeline is called the *MLOps pipeline*. The goal of this pipeline is the realization of continuous training, established in chapter *What is MLOps*. The general procedure is, that the MLOps pipeline is run in regular time intervals, where it will trigger the CD evaluation pipeline to check whether any CD has occurred. In case of CD, the training pipeline will be initiated, which will output a new model. This model will then replace the old model in the production environment (Figure 31).

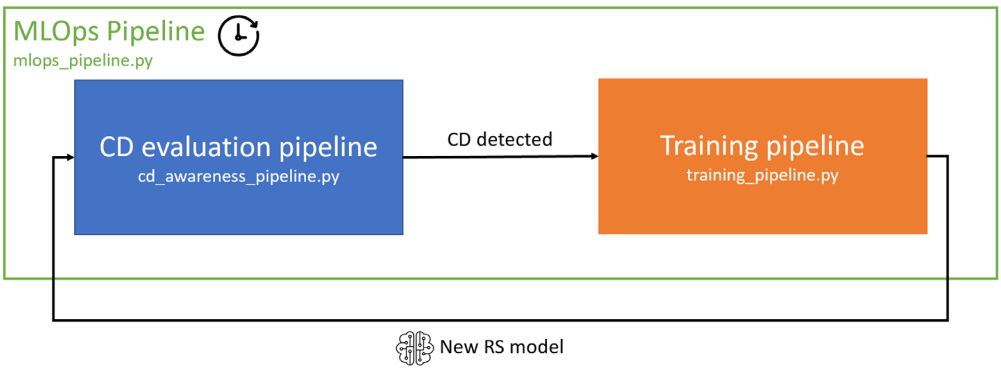


Figure 28: Outline of implemented MLOps pipeline

[13] CD evaluation pipeline and CD awareness pipeline are used synonymous in this thesis.

4.5.2 Training Pipeline

The TFX pipeline can be found in `training_pipeline.py` in the `airflow_pipelines` directory. Inside this file, a name for the pipeline is given. When the TFX pipeline is executed for the first time, a directory is created with the pipeline name, where all the outputs of the pipeline components are stored. In this project, the pipelines are located in `pipeline/pipelines`. The scope of this work only includes the standard TFX components. However, additional outputs are generated in form of the previously discussed PTAs. These are stored in a separate plots folder inside the pipeline directory. Since PTAs are executed within the training process inside the model file, they can technically be counted as part of the pipeline but aren't assigned to a dedicated TFX component. This pipeline follows the general structure of the TFX documentation, which starts with the ingestion of the training data.

CsvExampleGen. This component is used to read the training data. This component takes in a directory, which points to the location of the CSV file. In this case, the ingested data is the product of the data preparation task covered in chapter 4.2.3. Alternatively, a training and evaluation split can be specified. This artifact uses the default split of 2/3 training data and 1/3 evaluation data. When this component is executed, the CSV file is read and stored in TFRecord format in the `CsvExampleGen` directory (Figure 32).

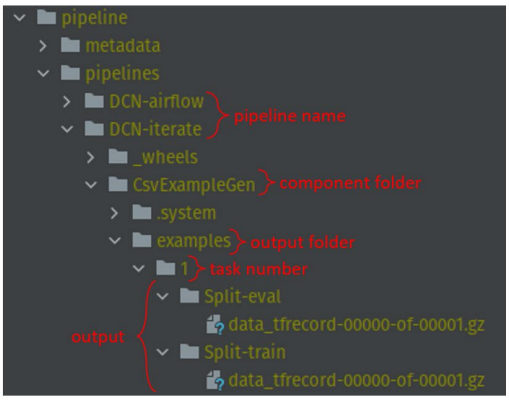


Figure 29: Folder Structure of TFX component[14]

StatisticsGen. StatisticsGen automatically generates statistics from the output of CsvExampleGen. This component doesn't require any further configuration. It returns a statistical overview of every feature in the dataset. The output of StatisticsGen is used by other components to infer the dataset schema and run anomaly detection. The output can also be visualized and be reviewed by humans. For the used MovieLens da-

[14] All components follow the same folder structure of `pipelines/{component_name}/{task_number}/{component_output}`. There reside the files that are passed on to the other components.

taset, StatisticsGen automatically highlights, that the gender distribution is biased towards the male gender (Figure 33)

SchemaGen. This component infers the composition of the data from the outputs of StatisticsGen and defines requirements for the data. These are read by proceeding components or can be manually reviewed by ML engineers. The output for the MovieLens dataset is seen in Figure 33. "*Type*" denotes the expected data type of the feature. "*Presence*" specifies, whether a feature is optional. "Valency" defines the number of values one feature example can hold. Features can also be assigned to a domain, which is useful to structure a feature store. For this artifact, all features are automatically ascribed the data type integer, because they got integer encoded in the data preparation step. No domains are given to the features, as no feature store is implemented in this MLOps system.

ExampleValidator. This component checks the training data for anomalies. As noted in the TensorFlow documentation, it automatically runs all necessary anomaly detection operations, without any configuration (Google LLC, 2021b). ExampleValidator calls skew and drift comparators from the TensorFlow Data Validator library and returns a protobuf file with the validation results. This component employs data distribution-based drift detection, instead of the error rate-based drift detection used in the monitoring part of the artifact. This means, that the CD analysis is directly done on the data. Running the data validation on the MovieLens dataset yields no detected anomalies (Figure 33).

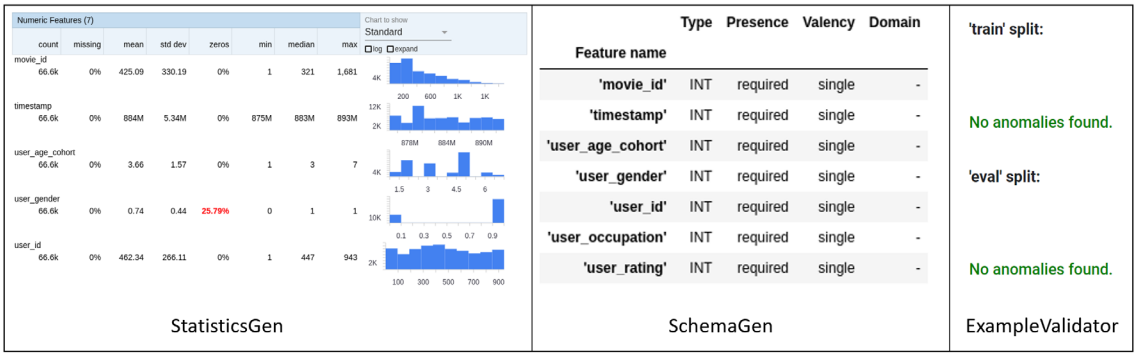


Figure 30: Outputs of StatisticsGen, SchemaGen & ExampleValidator

Trainer. The Trainer component is responsible for the execution of the model training task. In this implementation, the Trainer is given the model file mentioned in 4.3, training and evaluation steps as arguments and the training data from CsvExampleGen. Additionally, a custom parameter is used to pass in the destination path for the PTA plots. This component calls the `run_fn` function to execute the previously defined RS file.

Pusher. This component takes the newest trained model deploys it into a specified production environment. In this project, all models are pushed to the `serving_model` folder. The models are named after their deployment-timestamp, so the prediction service always chooses the most recent model to calculate ratings. This pusher does not

have conditions on what models to push. It naively pushes every model that gets trained on the pipeline.

ML Metadata. The history of all components is stored inside the ML Metadata SQLite database. This is a centralized space, where all TFX pipeline events are saved. In the default configuration used in this artifact, information such as TFX version, execution timestamps, output locations and component configurations are stored. This is done to provide an understanding of the pipeline process, to ensure reproducibility (Figure 34).

	artifact_id	name	is_custom_property	int_value	double_value	string_value	byte_value (UUID)
1	1	split_names	0	<null>	<null>	["train", "eval"]	<null>
2	1	name	1	<null>	<null>	DCN-iterate:2022-07-18T13:31:32.184064:CsvExampleGen:exa…	<null>
3	1	span	1	0	<null>	<null>	<null>
4	1	file_format	1	<null>	<null>	tfrecords_gzip	<null>
5	1	payload_format	1	<null>	<null>	FORMAT_TF_EXAMPLE	<null>
6	1	input_fingerprint	1	<null>	<null>	split:single_split,num_files:1,total_bytes:2617288,xor_c…	<null>
7	1	tfx_version	1	<null>	<null>	1.8.0	<null>
8	2	tfx_version	1	<null>	<null>	1.8.0	<null>
9	2	name	1	<null>	<null>	DCN-iterate:2022-07-18T13:31:32.184064:Trainer:model_run…	<null>
10	3	name	1	<null>	<null>	DCN-iterate:2022-07-18T13:31:32.184064:Trainer:model:0	<null>
11	3	tfx_version	1	<null>	<null>	1.8.0	<null>
12	4	pushed_destination	1	<null>	<null>	pipeline/serving_model/DCN-iterate/1658143931	<null>
13	4	pushed	1	1	<null>	<null>	<null>

Figure 31: Table in ML Metadata

The final pipeline is converted into an Airflow DAG with the `AirflowDagRunner` operator. This is done to ensure compatibility between the TFX pipeline and the overarching MLOps pipeline. In Figure 35, the visualization of the DAG inside the Airflow Web-UI can be seen.

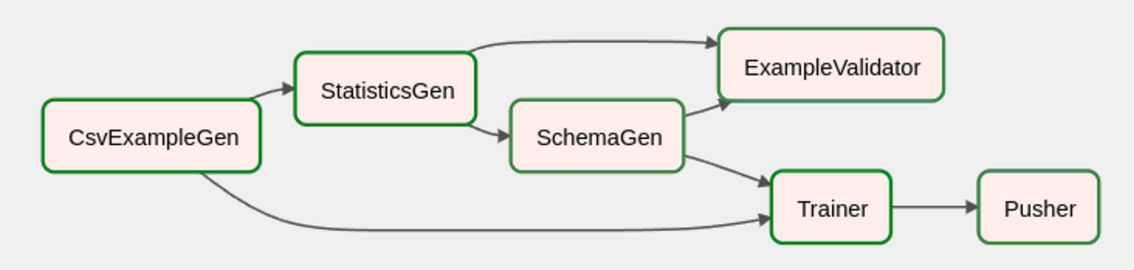


Figure 32: TFX DAG inside Airflow

4.5.3 CD Evaluation Pipeline

The inner workings of the operation of CD awareness have been described in the previous chapter. This segment is dedicated to explaining the incorporation of Airflow into this script. In order to implement the CD evaluation process into workflow, it needs to be wrapped inside a function, which can then be accessed by Airflow. Inside `cd_awareness_pipeline.py` the DAG is configured above the function, named `evaluate`. No schedule interval is defined, as this will be delegated to the MLOps pipeline. The entire script is defined as one task, as seen in Figure 36.

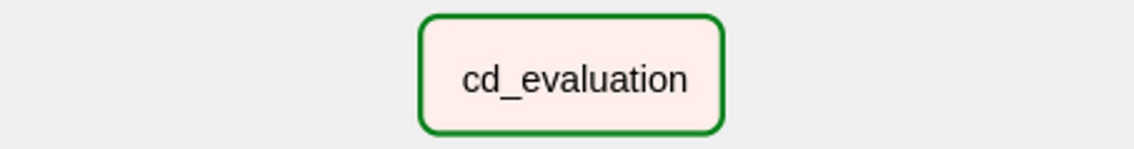


Figure 33: CD evaluation inside Airflow

4.5.4 MLOps Pipeline

In the following, it is explained how the CD awareness pipeline and the training pipeline are combined to form the final continuous training pipeline. The Airflow script is found in `mlops_pipeline.py`. First, a DAG context is created, which holds general pipeline information, such as the name and the execution schedule. The MLOps pipeline is run once a day, which is specified in the `schedule_interval` parameter with '@daily'[15]. For this pipeline, two Airflow operators are used to wrap both pipelines into a task: the `ShortCircuitOperator` and the `TriggerDagRunOperator`.

`ShortCircuitOperator` is used for the CD evaluation pipeline, which determines whether the following task (i.e., the training pipeline) should be run or skipped. It takes in the `evaluate` function in `cd_awareness_pipeline.py`, which return a Boolean. If `True` is returned (i.e., if CD has been detected), the retraining task will be executed. For this, `TriggerDagRunOperator` is used to run the TFX DAG outside the MLOps pipeline (Figure 37). These tasks are then orchestrated using the bitwise python operators at the end of the file.

Figure 34: MLOps pipeline inside Airflow

[15] '@daily' is a substitute for '0 0 * * *' in Cron notation

5 Evaluation

5.1 Recommender System

The recommender system forms one of the base requirements for this research (Table 1). In order to run an MLOps pipeline, a ML model needs to be present first. The objective of this requirement is the implementation of an SotA RS. In this chapter, it is evaluated, whether the used RS constitutes SotA. It is also evaluated to what extent the PTAs add to an MLOps system.

The RS of this artifact is a minimal implementation of Google's proposed DCN architecture, which employs cross-layers to leverage feature interactions more efficiently. No experimentation and optimization of the model architecture has been conducted, as model performance isn't considered in the context of this thesis. The cross-layer is the component, that makes the model constitute as SotA, as this technology has been successfully employed in large scale production environments (Wang et al., 2017). This project successfully incorporated a cross-layer into its RS, which is deemed sufficient for this prototype. Since this artifact only uses one cross-layer in its CN, it is limited to only learn feature interactions of the first degree. This means that this CN can only embed interactions between two features. In order to increase the degree of feature interactions, the CN can be expanded by additional cross-layers.

Reproducibility is a key aspect of MLOps. In order to make model training reproducible, the weights inside the DCN have been initialized with a seed. The feature embedding however still uses random weight initialization. While results of different training runs grew more similar with seeds, it could never be identical because of the embedding layer.

Post training actions have been introduced in this paper, in order to extract information about every trained RS model. The motive behind these small operations inside the training function, is to give additional insights into a model with the goal of improving model interpretability. Three PTAs have been implemented in this artifact: Model architecture visualization, TensorBoard logfiles and cross-layer visualization.

The visualization of the model architecture is considered useful as a first overview of the model. It provides general information about the composition of the model. For instance, it can be seen at first glance if the DCN follows a parallel or sequential architecture. Compared to the other PTAs, it is hard to make assumptions about a model's performance based off this visualization. It therefore serves more as a first overview to a model.

TensorBoard gives insights into the model training process. It maps training and evaluation metrics in a line chart. This visualizes the progression of a model during training. By plotting the training and evaluation loss, general assumptions about the performance of a trained model can be made. This is because loss values in ML training fol-

low a similar pattern. While the training loss approaches a value of zero with continuous training, the evaluation loss curve assumes a parabola-like shape, where it first decreases until it reaches a minimum and then increases again. This behavior can be seen in the TensorBoard visualization of an intentionally overfit RS model (Figure 23). From this plot it is easily inferred that the model is overfit and won't perform well in a production environment.

The cross-layer visualization leverages the learned weights of the cross network to highlight what cross feature interactions are deemed important by the model. CNs can give a unique insight into the model, which would otherwise not be possible with conventional NNs. If a CN is effectively learned, it can reveal correlations between features, which might not have been discovered otherwise. In the case of the model of this research, it also highlights the effects of overfitting, which are illustrated in Figure 38. This first plot shows the learned cross interaction of a CN after learning for 10 epochs, while the second plot shows the learned cross interactions after 200 epochs. In the left matrix, no prominent cross feature interactions are embedded into the CN. The second plot however clearly shows sensitivity towards the user and its demographic properties. This could indicate memorization of the users, which is a typical cause for overfit models. One limitation for cross-feature visualization is, that it becomes increasingly harder to visualize with larger degrees of cross-interactions.

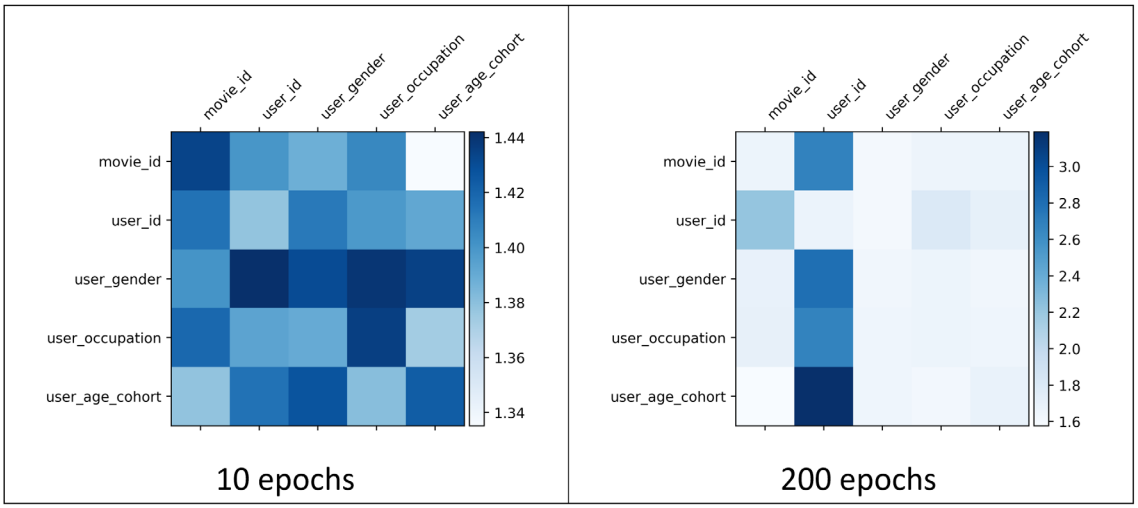


Figure 35: Learned cross-feature interactions for 10 and 200 epochs

All three PTAs together cover an entire model training process, from model configuration (i.e., model architecture) to model training and the trained model. These visualizations form a coherent and insightful overview of a model and could for example be integrated into a dashboard, as conceptualized in Figure 39.

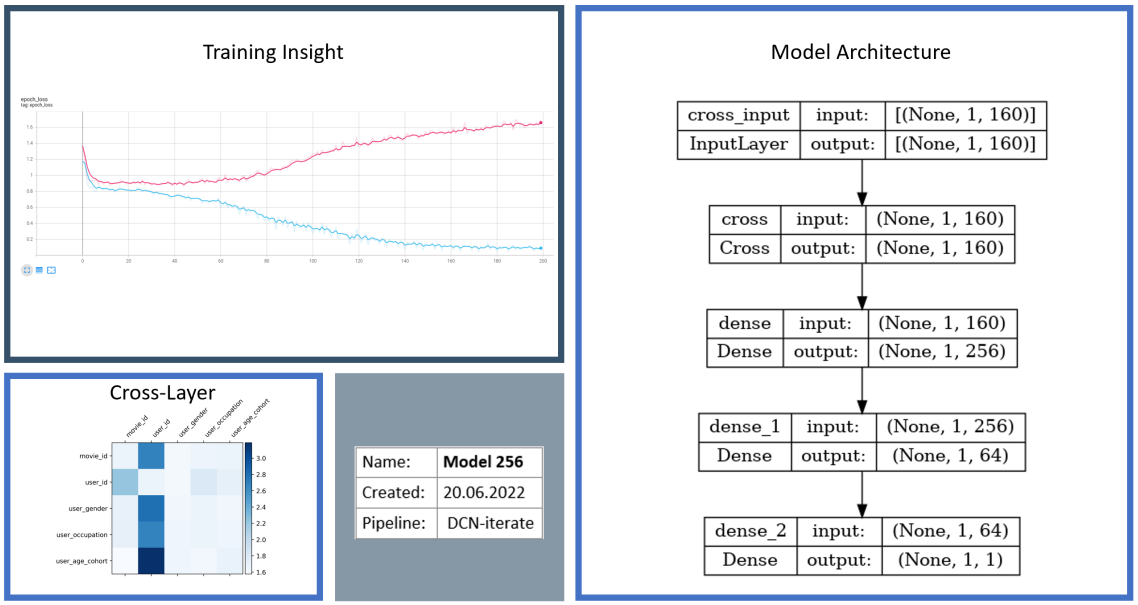


Figure 36: PTA dashboard concept

5.2 Concept Drift Awareness

CD awareness is part of the research requirement of this project (Table 1). As established earlier, CD awareness is composed of three parts, derived from Lu et al. (2018): CD detection, CD understanding and CD adaptation. Each of these three components are evaluated individually, in order to make an assessment of the CD awareness implementation of this artifact.

For CD detection, an error rate-based drift detection method is implemented, which infers the presence of CD from the performance of the deployed ML model. The data is grouped by a defined time interval and the RMSE value is calculated for each grouping (i.e., time window). During the CD detection run, the program considers the absolute RMSE value and the increase in RMSE from one time window to the next time window. Are these thresholds surpassed, the program will declare that CD has occurred. In test runs, the CD detection was successful and could correctly detect degrading model performance. The test was done with a grouping set by year. The timeline of the dataset only spans over two years, which means that the aggregate RMSE table the CD detection is run on, only contains two rows (Figure 27). An aggregation by month would increase the total amount of time windows but would also make the CD detection more susceptible to noise. In Figure 30 it is seen that the RMSE value can significantly shift on a month-by-month basis. This implementation doesn't effectively take noise into account. A more robust method for noise would be a rolling time window over multiple intervals.

CD understanding is covered by two implementations within the CD awareness component. The first is a byproduct of the CD detection function, which is a date of the time window CD was detected in. This is the simplest implementation of CD understanding. Consequently, the precision of the timestamp is bound by the aggregation level chosen

for the RMSE dataset. Therefore, choosing a grouping by year entails that the time of CD occurrence can only be pointed to the year it happened. The second implementation should give a more detailed insight into the CD at hand. A line chart of the RMSE values over time is used. For better visualization, a more granular aggregation is chosen. This artifact uses a grouping by month for this chart. Plots can reveal the behavior of CD in the data, which would otherwise be missing from a simple timestamp. For instance, it can be seen in Figure 30 that the error rate seems to follow a linear increase over time. Depending on how long the data is tracked, it could be observed which type of CD is present in the data. If the RMSE follows a repeating pattern for example, it could indicate a reoccurring concept. Based on this knowledge, more sophisticated CD compensation measures could be incorporated into the MLOps pipeline.

The CD adaptation requirement is realized within the MLOps pipeline evaluated in the next chapter.

5.3 Pipeline

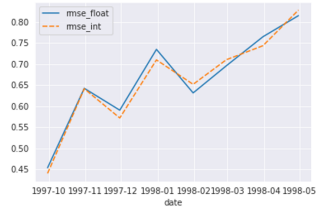
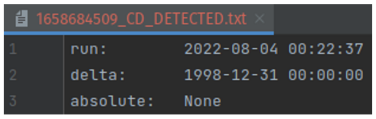
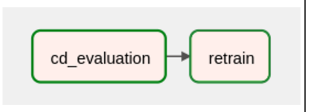
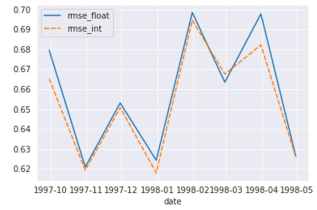
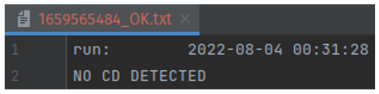
In order to evaluate the pipelines, a system test is conducted. This system test simulates a scenario, wherein CD is detected in the production environment, which triggers the TFX pipeline to train and deploy a new model. This concludes the first pipeline run. A second run is then initialized using the new model. This time, no CD should be detected. During this test, continuous training, CD adaptation and the training pipeline are covered, which includes all components for the pipeline evaluation.

For this system test, the starting RS model has only been trained on the first half of the dataset (i.e., the older half of the dataset). Furthermore, it has been purposely overfit using this data. The prediction service however is conducted with the complete dataset. This is done to artificially induce CD detection. The hypothesis behind this procedure is, that the model performs better on the first half of the dataset than on the second half, because the new data is unknown to the model. This anticipated drop in prediction quality is then used to simulate the performance drop due to CD. The second model is then trained with the complete dataset. Since the complete production dataset is now known to the model, there shouldn't be a spike in the RMSE anymore. This should prevent the CD detection function to classify CD.

In Table 4 the results of the system test can be seen. In the first pipeline run (old model) it is visible that the average error rate for the year 1997 is lower than the error rate of 1998. This RMSE difference superseded the defined threshold in the CD detection function, which is interpreted as CD. The timestamp for CD detection was set to the last day of 1998. This is because the CD detection function aggregates the dataset by year. The delta date in the `CD_DETECTED.txt` file therefore means that CD has occurred in the year 1998. Now that CD has been detected, the TFX pipeline is triggered to train the new model. Running the MLOps pipeline once more with the monitoring data of the new model shows that this time, no CD has been found. No CD means that

the training pipeline will be skipped, visualized in Airflow by the pink-outlined task (Table 4).

Table 4: Results of the system test

	RMSE on production data	Result of CD detection	Pipeline run
Old model			
Retrained model			

In this system test, the pipeline successfully executed the continuous training process and reacted as expected to the simulated CD. Upon execution, the pipeline could complete all tasks in one go. This means that the set requirement “integrated pipeline” is satisfied. The “CD adaptation” requirement is met as well, as it successfully initiates model retraining and was able to output an adjusted model. While the MLOps pipeline is configured to run without human intervention, the data ingested by the TFX pipeline isn’t automatically updated with new training data. Consequently, the model gets trained on the same data as the old model, as long as the training data isn’t manually swapped. Without human intervention the model can therefore not be adapted to the new concepts occurring in the production environment. For this reason, the requirement of “automated pipeline” isn’t achieved with this artifact.

The training pipeline employs 6 out of the 10 standard components offered by the TFX framework. The baseline components that are necessary to realize a training pipeline are CsvExampleGen, Trainer and Pusher. These are used in this artifact, such that the base requirement “base pipeline” is fulfilled. Furthermore, the following data analysis components are employed: StatisticsGen, SchemaGen and ExampleValidator. These make use of external TensorFlow libraries to run various data checks, without extensive configuration. These components provide additional information of the data and act as an additional safety mechanism in the process. This pipeline lacks a sophisticated deployment procedure, meaning that every model that gets trained, is pushed to production without further evaluation. This includes the danger that inferior models can be deployed and worsen the performance of the prediction service. Model quality prior to deployment could be ensured with the Evaluator and InfraValidator components. Data preprocessing is done inside the model file (i.e. feature embedding) and outside of the TFX pipeline (chapter 4.2.3). Ideally, these operations should be centralized in-

side the Transform component. Lasty, model optimization is not present in this artifact, as model performance isn't considered for this research.

The developed artifact can be classified as a level 1 maturity pipeline, according to Google's proposed MLOps maturity model. The central practice of this level is continuous training, which is the main implementation of this research project. Other characteristics of level 1 maturity are monitoring of the prediction service and modular components. Monitoring is realized with the CD awareness component of this artifact. Modularity is the core concept behind TFX and its components, but it is also employed outside the TFX pipeline. For instance, the strict separation of production and training pipeline ensures that both components can be developed isolated from each other. However, level 1 maturity isn't met throughout the whole MLOps system. The aforementioned manual updating of training data and the lack of compartmentalization of the data transformation process would be isolated instances of that.

6 Conclusion

This research concludes with an artifact that meets all but one of the specified requirements. The result is a CT pipeline that automatically detects and reacts to CD. This artifact serves as a demonstration of the benefits of MLOps. One goal of MLOps is to separate the development from the production process. With a CT pipeline, the lifecycle of a model can now be decoupled from the model development process. In the context of this artifact it means, that once a model file is integrated into the TFX pipeline, it will continuously and automatically adapt to changes in the data. The model training, as well as the decision of when to train a model, is therefore delegated to the pipeline and doesn't require any further attention from the ML engineer.

The TFX Trainer component imposes very few restrictions on the model, as long it is written in TF or Keras. Since the model file is treated as a black box by the Trainer, ML engineers have the flexibility to develop any arbitrary model. This makes the implementation of the SotA DCN recommender system possible, without having to rely on official support from TFX.

Information content is enriched through collection and storage of a variety of metadata, including PTAs. This creates transparency in the ML lifecycle and provides ML engineers with the necessary information to better understand the model, the data, or the ML system at hand. These insights can then be incorporated into the development process, where improvements to said components can be made.

Furthermore, the modular structure of this artifact serves as an aid during development. Components can be developed independently from each other and later be integrated into the pipeline. In combination with Airflow, pipeline errors are visualized inside the Web-UI, where they can be traced back to their respective task. The log files can then be used for further error-understanding. Modularization has the benefit that problems can be attributed to specific components. This simplifies the debugging process during development.

Consequently, a modular structure allows for rapid iterations of the ML system, which is another goal set out by MLOps. The result of this research project can be seen as a baseline, on which can be expanded in future developments. In order to satisfy the remaining requirement, a third pipeline would have to be implemented, which would automatically update the training data for the TFX pipeline. This would make the artifact fully autonomous. At the same time, data preparation should be merged into one component and the components for model quality checks should be added.

Retrospectively, the pipeline evaluation approach used in this research is found to be suboptimal and should be revised in future work. It serves as a demonstration of the capabilities of the pipeline, although in doing so it violates established ML conventions in order to simulate a CD scenario. These violations are caused by the use of training

data in the test environment, as well as the intentional overfit of the RS. A more elegant solution would have been to use a dedicated CD dataset or generate artificial CD examples from the MovieLens dataset.

The next evolution of this artifact would be the transition from a ML pipeline automation (i.e., level 1 maturity) to a CI/CD pipeline automation (i.e., level 2 maturity). Level 2 maturity envisions the operationalization of the complete end-to-end ML lifecycle, from pipeline experimentation to pipeline development and monitoring. This requires, among other things, a completely automated test regiment for pipeline deployment, which is lacking in the current iteration of this artifact. Future research could consist of outlining a process to migrate the current pipeline to level 2 maturity.

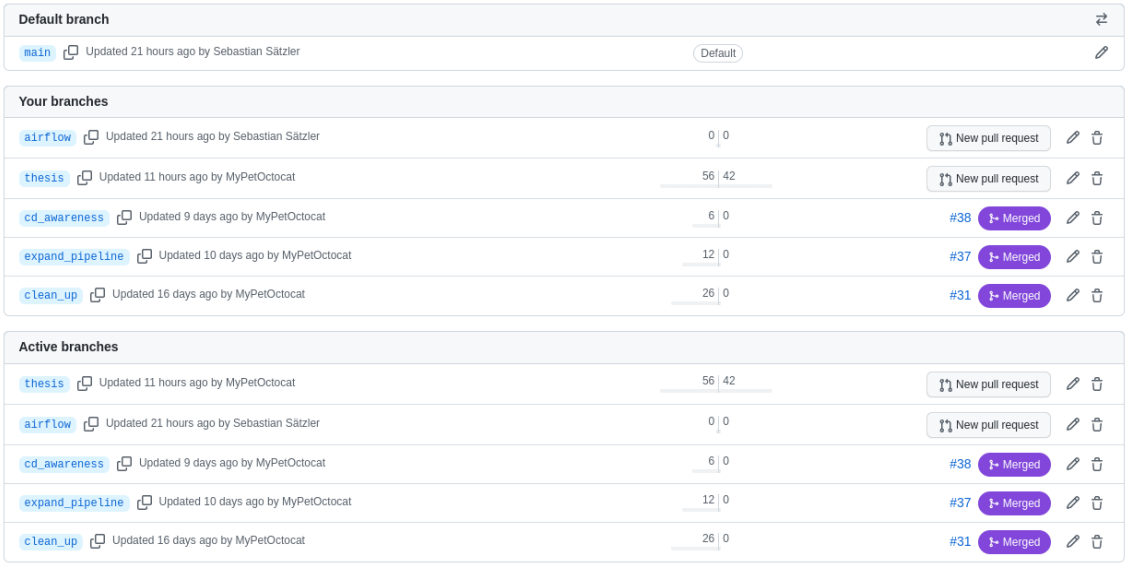
MLOps is a novel field that still lacks a universally agreed upon definition and understanding. While the necessity for MLOps is generally acknowledged by the parties involved in the world of ML, there are different ideas of what actually constitutes MLOps and how MLOps is most effectively implemented. This leads to a vast landscape of different proposals and software solutions. As seen with this artifact, a complete MLOps pipeline is hardly covered by only one software solution alone. This is, among other things, due to the wide range of unique ML problems, which is an obstacle that all-in-one ML solutions face. For this reason, platforms like KubeFlow are growing in popularity. Instead of offering one software solution alone, it provides a centralized environment with a plethora of open-source software to conduct large-scale MLOps on.

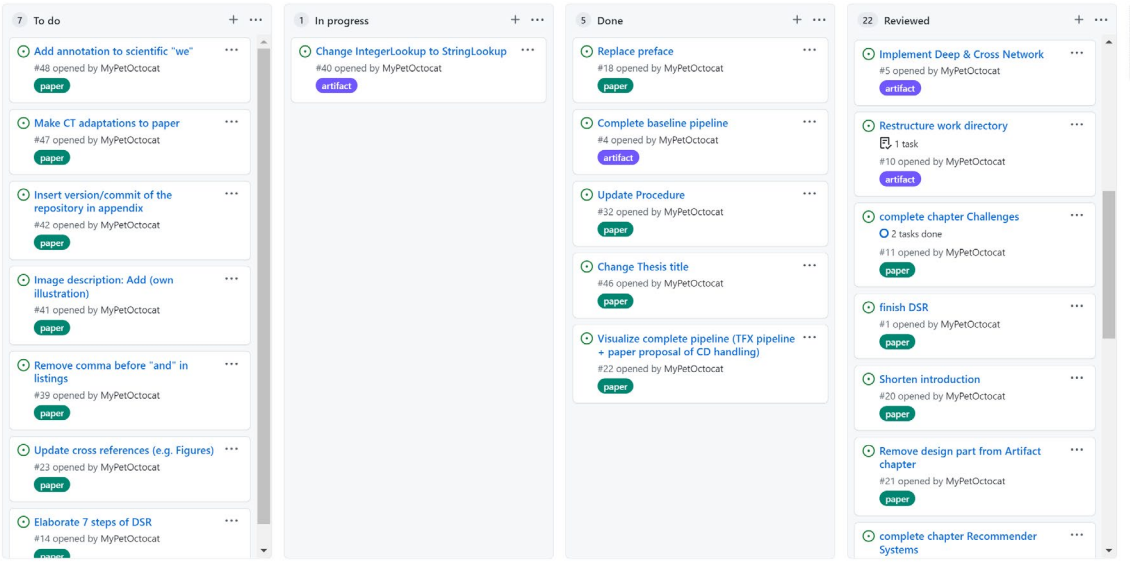
As long as compatibility between the different ML tools is ensured, it is possible to efficiently compose complex MLOps systems using multiple software packages. This research directly benefited from the compatibility of Airflow and TFX for instance. It streamlined the creation of the artifact and reduced development time. In the end, both software solutions complemented each other to form a more coherent final product.

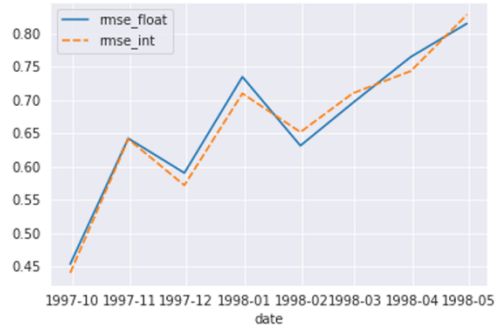
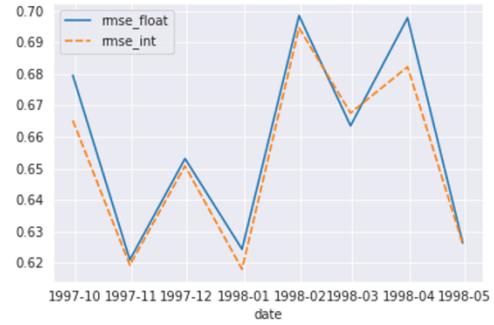
It remains to be seen in which direction MLOps will develop over the next years, but with continuing ML adoption in the industry sector, it is destined to grow in relevancy. Free open-source software, like the ones used in this research project, makes ML more tangible for real-world applications, which could further open the doors for wider ML adoption.

Appendix:

A.1 Figures





	RMSE on production data	Result of CD detection	Pipeline run
Old model		1658684509_CD_DETECTED.txt 1 run: 2022-08-04 00:22:37 2 delta: 1998-12-31 00:00:00 3 absolute: None	
Retrained model		1659565484_OK.txt 1 run: 2022-08-04 00:31:28 2 NO CD DETECTED	

A.2 Source Code

The printed version of this bachelor thesis contains a CD with the source code of this artifact. Alternatively the source code is available online at https://github.com/MyPetOctocat/bachelor_2022.

airflow_pipelines

cd_eval_functions/cd_detector.py

```
# Detects CD in the dataset based on threshold parameters

# Scans Dataframe for threshold violation
def cd_detection(rmse_df, threshold):
    for index, elem in rmse_df.iterrows():
        if (elem > threshold).all():
            return False, index  # If CD detected return Bool + Date window of detection
    return True, None  # If no CD detection found, return Bool and None as placeholder

# Receives DF of RMSE values and returns a Bool: True --> Data is OK | False --> Datadrift detected
def cd_detector(rmse_df, delta_threshold, absolute_threshold):
    """
    delta_threshold: Acceptable RMSE increase from one window to another
    absolute_threshold: Threshold RMSE value that applies to every value
    """
    rmse_delta = rmse_df.rolling(2).apply(
        lambda x: x.iloc[1] - x.iloc[0])  # Calculate delta between 2 values within a rolling window

    # Checks whether RMSE thresholds have been surpassed.
    delta_rmse_ok = cd_detection(rmse_delta, delta_threshold)
    absolute_rmse_ok = cd_detection(rmse_df, absolute_threshold)

    # Returns tuple: 1. index is bool, 2. index is list of CD timestamp for delta and absolute
threshold
    # Info to bool: True if both absolute and delta are within threshold, else False is passed
    return delta_rmse_ok[0] and absolute_rmse_ok[0], [delta_rmse_ok[1], absolute_rmse_ok[1]]
```

cd_eval_functions/rmse_calculator.py

```
# Calculates Root Mean Squared Error for a DF and returns Series
import pandas as pd
import numpy as np


# Define RMSE function (Metric for CD evaluation)
def rmse(predictions, targets):
    return np.sqrt(((predictions - targets) ** 2).mean())


# Calculates RMSE for entire DF grouped by date and returns Series (freq: "Y" -> Year, "M" -> Month,
"1W" -> Week)
def rmse_calc(df, freq):
    return df.groupby(pd.Grouper(key="date", freq=freq)).apply(lambda x: rmse(x.iloc[:, 1], x.iloc[:,
2]))
```

cd_eval_functions/rmse_calculator.py

```
# Generates and saves plot from RMSE DF (part of CD understanding)

import seaborn as sns


def plot_gen(rmse_df, location):
    rmse_trend = sns.lineplot(data=rmse_df)

    # Save plot
    fig = rmse_trend.get_figure()
    fig.savefig(location)
    return fig
```

cd_awareness_pipeline.py

```
from airflow import DAG
from airflow.operators.python import PythonOperator

import pandas as pd
from datetime import datetime
import os
```

```
import glob

# switch to production directory
os.chdir("/home/cory/PycharmProjects/bachelor_2022/artifact/production/monitoring")
print(os.getcwd())

# import functions
from cd_eval_functions.rmse_calculator import rmse_calc
from cd_eval_functions.cd_detector import cd_detector
from cd_eval_functions.rmse_plot_generator import plot_gen as rpg

# Set up Airflow
with DAG("cd_evaluation",  # Dag id
    start_date=datetime(2022, 8, 3),  # start date, the 1st of January 2021
    schedule_interval=None,  # This pipeline should only be externally triggered (e.g. by MLOps Pipe-
line)
    catchup=False  # Catchup
) as dag:

    def evaluate():

        # load Dataframe with monitoring data of newest model
        PIPELINE_NAME = 'DCN-iterate'
        SERVING_MODEL_DIR = os.path.join('../../pipeline/serving_model', PIPELINE_NAME)

        model_name = max(glob.glob(os.path.join(SERVING_MODEL_DIR, '*/')),
key=os.path.getmtime).split("/")[-2]

        df = pd.read_csv(f'{model_name}_monitoring.csv')  # Load prediction score dataset

        df['date'] = pd.to_datetime(df['timestamp'], unit='s')

        # Group by date and calculate the rmse for each month/week/year & join them to one dataset
        group_by = "Y"

        rmse_float = rmse_calc(df[['date', 'user_rating', 'pred_rating']], group_by)  # RMSE of na-
tive predictions
        rmse_int = rmse_calc(df[['date', 'user_rating', 'pred_int_rating']], group_by)  # RMSE of
rounded predictions
        rmse_df = pd.DataFrame(dict(rmse_float=rmse_float, rmse_int=rmse_int))

        # CD detection (determine whether CD occurred in the data)
        no_cd = cd_detector(rmse_df, delta_threshold=0.1, absolute_threshold=1.5)

        # Write file of the CD evaluation result (prerequisite to CD adaptation: File can be read by
Airflow an initiate TFX retraining)
        if no_cd[0]:
            with open(f'evaluation_outputs/{model_name}_OK.txt', 'w') as file:
                file.write("run:\t\t" + str(datetime.now()) + "\nNO CD DETECTED")
        else:
            with open(f'evaluation_outputs/{model_name}_CD_DETECTED.txt', 'w') as file:
                file.write("run:\t\t{}\ndelta:\t\t{}\nabsolute:\t{}".format(datetime.now(),
no_cd[1][0], no_cd[1][1]))

        # CD understanding (give visual feedback about CD)
        group_by = "M"  # Group by month for better visualization

        vis_rmse_float = rmse_calc(df[['date', 'user_rating', 'pred_rating']], group_by)  # RMSE of
native predictions
        vis_rmse_int = rmse_calc(df[['date', 'user_rating', 'pred_int_rating']],
                                 group_by)  # RMSE of rounded predictions
        vis_rmse_df = pd.DataFrame(dict(rmse_float=vis_rmse_float, rmse_int=vis_rmse_int))

        rpg(vis_rmse_df, f'evaluation_outputs/{model_name}_rmse_trend.png')

        return not no_cd[0]  # Is passed on to ShortCircuitOperator: If True (CD has been detected)
trigger TFX training pipeline

    cd_evaluation = PythonOperator(
        task_id='cd_evaluation',
        python_callable=evaluate,
        dag=dag
    ))
    return fig
```

mlops_pipeline.py

```
from airflow import DAG
from airflow.operators.python import ShortCircuitOperator
from datetime import datetime
from airflow.operators.trigger_dagrun import TriggerDagRunOperator

from cd_awareness_pipeline import evaluate

with DAG("mlops_pipeline",  # Dag id
    start_date=datetime(2022, 8, 3),  # start date, the 1st of January 2021
    schedule_interval='@daily',  # Cron expression, here it is a preset of Airflow, @daily means once
every day.
    catchup=False  # Catchup
```

```
) as dag:

    # Externally trigger individual pipelines
    cd_eval = ShortCircuitOperator(  # This operator determines whether the next DAG (retraining)
will be executed
        task_id='cd_evaluation',
        python_callable=evaluate,
        dag=dag)

    train = TriggerDagRunOperator(  # Trigger TFX pipeline
        task_id='retrain',
        trigger_dag_id='DCN-airflow',
        dag=dag
    )

    cd_eval >> train)
    return fig
```

training_pipeline.py

```
from tfx import v1 as tfx
from tfx.orchestration.airflow.airflow_dag_runner import AirflowDagRunner
from tfx.orchestration.airflow.airflow_dag_runner import AirflowPipelineConfig


from tfx.orchestration import data_types
from tfx.orchestration import metadata
from tfx.orchestration import pipeline

import os
#from datetime import datetime
import datetime

import tensorflow_model_analysis as tfma
from typing import List

os.chdir("/home/cory/PycharmProjects/bachelor_2022/artifact") # change to working_dir

PIPELINE_NAME = 'DCN-airflow'
#WORKING_DIR = 'bachelor_2022/artifact'
PIPE_DIR = 'pipeline'

# Directory where MovieLens 100K rating data resides
DATA_ROOT = os.path.join('data', PIPELINE_NAME)
print(DATA_ROOT)
# Output directory to store artifacts generated from the pipeline.
PIPELINE_ROOT = os.path.join(PIPE_DIR, 'pipelines', PIPELINE_NAME)
print(PIPELINE_ROOT)
# Path to a SQLite DB file to use as an MLMD storage.
METADATA_PATH = os.path.join(PIPE_DIR, 'metadata', PIPELINE_NAME, 'metadata.db')
print(METADATA_PATH)
# Output directory where created models from the pipeline will be exported.
SERVING_MODEL_DIR = os.path.join(PIPE_DIR, 'serving_model', PIPELINE_NAME)
print(SERVING_MODEL_DIR)

MODEL_PLOTS = os.path.join(PIPE_DIR, 'pipelines', PIPELINE_NAME, 'plots')
print(MODEL_PLOTS)

from absl import logging
logging.set_verbosity(logging.INFO)  # Set default logging level.

_trainer_module_file = 'model_source/dcn_ranking_training.py'

_beam_pipeline_args = [
    '--direct_running_mode=multi_processing',
    '--direct_num_workers=0',
]

# Airflow-specific configs; these will be passed directly to airflow
_airflow_config = {
    'schedule_interval': None,
    'start_date': datetime.datetime(2022, 8, 3),
}



def _create_pipeline(pipeline_name: str, pipeline_root: str, data_root: str,
                     module_file: str, serving_model_dir: str,
                     metadata_path: str, plot_path: str, beam_pipeline_args: List[str]) ->
tfx.dsl.Pipeline:


  # Brings data into the pipeline.
  example_gen = tfx.components.CsvExampleGen(input_base=data_root)

  # Evaluate rudimentary statistics on the dataset
  statistics_gen = tfx.components.StatisticsGen(
      examples=example_gen.outputs['examples']
      )
```

```
  # Read and infer schema of the dataset
  schema_gen = tfx.components.SchemaGen(
    statistics=statistics_gen.outputs['statistics']
  )

  # Check dataset for anomalies
  example_validator = tfx.components.ExampleValidator(
    statistics=statistics_gen.outputs['statistics'],
    schema=schema_gen.outputs['schema'])

  # Uses user-provided Python function that trains a model.
  trainer = tfx.components.Trainer(
      module_file=module_file,
      examples=example_gen.outputs['examples'],
      schema=schema_gen.outputs['schema'],
      train_args=tfx.proto.TrainArgs(num_steps=12),
      eval_args=tfx.proto.EvalArgs(num_steps=24),
      custom_config={"plot_path": plot_path})

  # Pushes the model to a filesystem destination.
  pusher = tfx.components.Pusher(
      model=trainer.outputs['model'],
      push_destination=tfx.proto.PushDestination(
          filesystem=tfx.proto.PushDestination.Filesystem(
              base_directory=serving_model_dir)))

  # Following three components will be included in the pipeline.
  components = [
      example_gen,
      statistics_gen,
      schema_gen,
      example_validator,
      trainer,
      pusher,
  ]

  return tfx.dsl.Pipeline(
      pipeline_name=pipeline_name,
      pipeline_root=pipeline_root,
      metadata_connection_config=tfx.orchestration.metadata
      .sqlite_metadata_connection_config(metadata_path),
      components=components,
      beam_pipeline_args=beam_pipeline_args
  )

DAG = AirflowDagRunner((AirflowPipelineConfig(_airflow_config))).run(
  _create_pipeline(
      pipeline_name=PIPELINE_NAME,
      pipeline_root=PIPELINE_ROOT,
      data_root=DATA_ROOT,
      module_file=_trainer_module_file,
      serving_model_dir=SERVING_MODEL_DIR,
      metadata_path=METADATA_PATH,
      plot_path=MODEL_PLOTS,
      beam_pipeline_args=_beam_pipeline_args))
```

data_fetch

movielens_csv_generator.ipynb

```
import os
import pandas as pd

# Change work directory to /artifact
print(os.getcwd())
os.chdir("../data/downloads")
print(os.getcwd())

ds_name = 'ml-100k'

# Fetch data (only run for the first time)
# !wget https://files.grouplens.org/datasets/movielens/{ds_name}.zip
# !unzip {ds_name}.zip
# !rm {ds_name}.zip

# Instantiate all needed dataframes from download files

# Instantiate DF of user-movie rating
data_cols = ['user_id', 'movie_id', 'user_rating', 'timestamp']
data_df = pd.read_csv(f'{ds_name}/u.data', sep='\t', names=data_cols)
print(data_df.sample(1))

# Instantiate DF of user demography
user_cols = ['user_id', 'raw_user_age', 'user_gender', 'user_occupation_text', 'user_zip_code']
user_df = pd.read_csv(f'{ds_name}/u.user', sep='|', names=user_cols)
```

```
print(user_df.sample(1))

movie_cols = ['movie_id', 'movie_name', 'release_date', ' ','link', 'genre1', 'genre2', 'genre3',
'genre4', 'genre5', 'genre6', 'genre7', 'genre8', 'genre9', 'genre10', 'genre11', 'genre12', 'gen-
re13', 'genre14', 'genre15', 'genre16', 'genre17', 'genre18', 'genre19', 'genre20']
movie_df = pd.read_csv(f'{ds_name}/u.item', sep='|', names=movie_cols, encoding = "ISO-8859-1")
# keep columns movie_id, movie_name, release_date
movie_df = movie_df.iloc[:, :3]
movie_df.sample(1)

# Merge individual dfs into one

# Augment data_df with user_df
df = data_df.merge(user_df, on='user_id')
# Augment df with movie_df
df = df.merge(movie_df, on='movie_id')
df

# Simple data augmentation

# Bucketize user age (as seen in tfds)
def bucketizer(age):
    if age < 18:
        return 1
    elif age >= 18 and age < 25:
        return 18
    elif age >= 25 and age < 35:
        return 25
    elif age >= 35 and age < 45:
        return 35
    elif age >= 45 and age < 50:
        return 45
    elif age >= 50 and age < 55:
        return 50
    else:
        return 56

df['bucketized_user_age'] = df['raw_user_age'].apply(bucketizer)

# Convert gender to bool value
df['bool_user_gender'] = df['user_gender'].map({'M': True, 'F': False})
df

df['int_user_gender'] = df['bool_user_gender'].astype(int)

# Extract year as int from release date
# def year_extractor(date):
#     return date.split('-')[2]
#
# df['release_year'] = df['release_date'].apply(year_extractor)
Df

# Create a ready-to-user df for the model
df_processed = df[['user_id', 'movie_id', 'user_rating', 'timestamp', 'user_occupation_text', 'buck-
etized_user_age', 'int_user_gender']]

df_processed['user_occupation_text'], _ = pd.factorize(df_processed['user_occupation_text'])
df_processed['bucketized_user_age'], _ = pd.factorize(df_processed['bucketized_user_age'])

df_processed = df_processed.rename(columns={"user_occupation_text" : "user_occupation", "bucket-
ized_user_age" : "user_age_cohort", "int_user_gender" : "user_gender"})
df_processed['user_occupation'] = df_processed['user_occupation'].apply(lambda x: x+1)
df_processed['user_age_cohort'] = df_processed['user_age_cohort'].apply(lambda x: x+1)

df_processed

# Save complete file in the data repository
os.chdir("../recommender-datasets/")
df.to_csv(f"{ds_name}_augmented.csv", index=False)
print(df.shape)


# Split data in half by timestamp (for concept drift analysis)
median_timestamp = df['timestamp'].median()

df_1 = df.loc[df['timestamp'] <= median_timestamp]
df_2 = df.loc[df['timestamp'] > median_timestamp]

# Check split
print(df_1.shape)
print(df_2.shape)

df_1.to_csv(f"{ds_name}_part1_augmented.csv", index=False)
df_2.to_csv(f"{ds_name}_part2_augmented.csv", index=False)

# Save processed file in the data repository

df_processed.to_csv(f"{ds_name}_ready.csv", index=False)
print(df_processed.shape)
```

```
df_processed_1 = df_processed.loc[df_processed['timestamp'] <= median_timestamp]
df_processed_2 = df_processed.loc[df_processed['timestamp'] > median_timestamp]

# Check split
print(df_processed_1.shape)
print(df_processed_2.shape)

df_processed_1.to_csv(f"{ds_name}_ready_part1.csv", index=False)
df_processed_2.to_csv(f"{ds_name}_ready_part2.csv", index=False)
```

model_source

dcn_ranking_training.py

```
from typing import Dict, Text
from typing import List
from pathlib import Path

import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import make_axes_locatable

import numpy as np
import tensorflow as tf

from tensorflow_metadata.proto.v0 import schema_pb2
import tensorflow_recommenders as tfrs
from tensorflow_transform.tf_metadata import schema_utils
from tfx import v1 as tfx
from tfx_bsl.public import tfxio

_FEATURE_KEYS = ["movie_id","user_id","user_gender", "user_occupation", "user_age_cohort"]
_LABEL_KEY = 'user_rating'

_FEATURE_SPEC = {
    **{
        feature: tf.io.FixedLenFeature(shape=[1], dtype=tf.int64)
        for feature in _FEATURE_KEYS
    }, _LABEL_KEY: tf.io.FixedLenFeature(shape=[1], dtype=tf.int64)
}

class RankingModel(tf.keras.Model):

  def __init__(self):
    super().__init__()

    # Define the dimension the feature values should be embedded in
    embedding_dimension = 32
    self.embedding_dims = embedding_dimension
    # Create np array with incrementing values as the vocabulary
    unique_user_ids = np.array(range(943)).astype(str)
    unique_movie_ids = np.array(range(1682)).astype(str)
    unique_occupation_ids = np.array(range(21)).astype(str)
    unique_gender_ids = np.array(range(2)).astype(str)
    unique_age_ids = np.array(range(7)).astype(str)


    ## String values embeddings
    # Compute embeddings for users.
    self.user_embeddings = tf.keras.Sequential([
        tf.keras.layers.Input(shape=(1,), name='user_id', dtype=tf.int64),
        tf.keras.layers.Lambda(lambda x: tf.as_string(x)),
        tf.keras.layers.StringLookup(
            vocabulary=unique_user_ids, mask_token=None),
        # Create embedding layer of dimension 943x32
        tf.keras.layers.Embedding(
            len(unique_user_ids) + 1, embedding_dimension)
    ])

    # Compute embeddings for movies.
    self.movie_embeddings = tf.keras.Sequential([
        tf.keras.layers.Input(shape=(1,), name='movie_id', dtype=tf.int64),
        tf.keras.layers.Lambda(lambda x: tf.as_string(x)),
        tf.keras.layers.StringLookup(
            vocabulary=unique_movie_ids, mask_token=None),
        tf.keras.layers.Embedding(
            len(unique_movie_ids) + 1, embedding_dimension)
    ])

    # Compute embeddings for occupations.
    self.occupation_embeddings = tf.keras.Sequential([
        tf.keras.layers.Input(shape=(1,), name='user_occupation', dtype=tf.int64),
        tf.keras.layers.Lambda(lambda x: tf.as_string(x)),
        tf.keras.layers.StringLookup(
            vocabulary=unique_occupation_ids, mask_token=None),
```

```
        tf.keras.layers.Embedding(
            len(unique_occupation_ids) + 1, embedding_dimension)
    ])

    ## Int value embeddings
    # Compute embeddings for gender.
    self.gender_embeddings = tf.keras.Sequential([
        tf.keras.layers.Input(shape=(1,), name='user_gender', dtype=tf.int64),
        tf.keras.layers.IntegerLookup(
            vocabulary=unique_gender_ids, mask_token=None),
        tf.keras.layers.Embedding(
            len(unique_gender_ids) + 1, embedding_dimension)
    ])

    # Compute embeddings for age.
    self.age_embeddings = tf.keras.Sequential([
        tf.keras.layers.Input(shape=(1,), name='user_age_cohort', dtype=tf.int64),
        tf.keras.layers.IntegerLookup(
            vocabulary=unique_age_ids, mask_token=None),
        tf.keras.layers.Embedding(
            len(unique_age_ids) + 1, embedding_dimension)
    ])

    # Cross Layer
    self.cross_layer =
tfrs.layers.dcn.Cross(kernel_initializer=tf.keras.initializers.RandomNormal(seed=1)) # Use seeds to
make model reproducible

    # Compute predictions.
    self.ratings = tf.keras.Sequential([
        self.cross_layer,
        tf.keras.layers.Dense(256, activation='relu', ker-
nel_initializer=tf.keras.initializers.RandomNormal(seed=1)),
        tf.keras.layers.Dense(64, activation='relu', ker-
nel_initializer=tf.keras.initializers.RandomNormal(seed=1)),
        tf.keras.layers.Dense(1, kernel_initializer=tf.keras.initializers.RandomNormal(seed=1))
    ])

  def call(self, inputs):

    user_id, movie_id, user_gender, user_occupation, user_age = inputs

    # Calculate embedding for each feature and save in *_embedding variable
    user_embedding = self.user_embeddings(user_id)
    movie_embedding = self.movie_embeddings(movie_id)
    gender_embedding = self.gender_embeddings(user_gender)
    occupation_embedding = self.occupation_embeddings(user_occupation)
    age_embedding = self.age_embeddings(user_age)


    # Create embedding layer
    return self.ratings(tf.concat([user_embedding, movie_embedding, gender_embedding, occupa-
tion_embedding, age_embedding], axis=2))


class MovielensModel(tfrs.models.Model):

  def __init__(self):
    super().__init__()
    self.ranking_model: tf.keras.Model = RankingModel()
    self.task: tf.keras.layers.Layer = tfrs.tasks.Ranking(
        loss=tf.keras.losses.MeanSquaredError(),
        metrics=[tf.keras.metrics.RootMeanSquaredError()])

  def call(self, features: Dict[str, tf.Tensor]) -> tf.Tensor:
    return self.ranking_model((features['user_id'], features['movie_id'], features['user_gender'],
features['user_occupation'], features['user_age_cohort']))

  def compute_loss(self,
                   features: Dict[Text, tf.Tensor],
                   training=False) -> tf.Tensor:

    labels = features[1]
    rating_predictions = self(features[0])

    # The task computes the loss and the metrics.
    return self.task(labels=labels, predictions=rating_predictions)


def _input_fn(file_pattern: List[str],
              data_accessor: tfx.components.DataAccessor,
              schema: schema_pb2.Schema,
              batch_size: int = 256) -> tf.data.Dataset:
  return data_accessor.tf_dataset_factory(
      file_pattern,
      tfxio.TensorFlowDatasetOptions(
          batch_size=batch_size, label_key=_LABEL_KEY),
      schema=schema).repeat()
```

```
def _build_keras_model() -> tf.keras.Model:
  return MovielensModel()


# TFX Trainer will call this function.
def run_fn(fn_args: tfx.components.FnArgs):
  """Train the model based on given args.

  Args:
    fn_args: Holds args used to train the model as name/value pairs.
  """

  # Generate training logfiles for tensorboard
  from datetime import datetime
  logdir = "pipeline/pipelines/DCN-iterate/logs/scalars/" + datetime.now().strftime("%Y%m%d-%H%M%S")
  tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=logdir)

  # Derive data schema from generated _FEATURE_SPEC dictionary
  schema = schema_utils.schema_from_feature_spec(_FEATURE_SPEC)

  train_dataset = _input_fn(
      fn_args.train_files, fn_args.data_accessor, schema, batch_size=8192)
  eval_dataset = _input_fn(
      fn_args.eval_files, fn_args.data_accessor, schema, batch_size=4096)

  model = _build_keras_model()

  model.compile(optimizer=tf.keras.optimizers.Adagrad(learning_rate=0.1))

  model.fit(
      train_dataset,
      steps_per_epoch=fn_args.train_steps,
      epochs=10,
      validation_data=eval_dataset,
      validation_steps=fn_args.eval_steps,
      callbacks=[tensorboard_callback])

  model.save(fn_args.serving_model_dir)


  ###  Display model summary
  print("\n#####################################")
  print(model.summary())
  print()

  ### Post-Training Actions

  # Save plot of model architecture
  model_num = fn_args.serving_model_dir.split("/")[-2]   # extract model number
  img_dir = fn_args.custom_config["plot_path"] + f"/{model_num}"
  print(img_dir)
  Path(img_dir).mkdir(parents=True, exist_ok=True)
  tf.keras.utils.plot_model(model.ranking_model.ratings,
to_file=f"{img_dir}/model_architecture_{model_num}.png", show_shapes=True)
  print()

  ### Cross feature Visualization
  mat = model.ranking_model.cross_layer._dense.kernel # Cross weights matrix
  features = _FEATURE_KEYS

  block_norm = np.ones([len(features), len(features)])
  dim = model.ranking_model.embedding_dims

  # Compute the norms of the blocks.
  for i in range(len(features)):
    for j in range(len(features)):
      # Norm of 32x32 Matrix is calculated | 32x32 values --> 1 value
      block = mat[i * dim:(i + 1) * dim,    # 32x32 blocks are retrieved from cross network
                  j * dim:(j + 1) * dim]
      block_norm[i,j] = np.linalg.norm(block, ord="fro") # Frobenius norm is used | norm of each
matrix element is calculated and added together
  # Create plot
  plt.figure(figsize=(9,9))
  im = plt.matshow(block_norm, cmap=plt.cm.Blues)
  ax = plt.gca()
  divider = make_axes_locatable(plt.gca())
  cax = divider.append_axes("right", size="5%", pad=0.05)
  plt.colorbar(im, cax=cax)
  cax.tick_params(labelsize=10)
  _ = ax.set_xticklabels([""] + features, rotation=45, ha="left", fontsize=10)
  _ = ax.set_yticklabels([""] + features, fontsize=10)

  plt.savefig(f"{img_dir}/cross_features_{model_num}", dpi=500, bbox_inches='tight')
```

production/prediction_service

batch_predictor.py

```
# Takes in a Dataframe and returns a new Dataframe with prediction values

from predictor import prediction_server as ps

def predict_batch(df, model):
    for index, row in df.iterrows():
        entry = row.to_dict()  # Dictionary representation of row for predictor
        pred = ps(model, entry)  # Rating Prediction
        df.at[index, 'pred_rating'] = pred  # set predicted rating

        # Convert float to integer for absolute ratings (conditions to keep rating interval [1;5])
        int_pred = int(round(pred))

        if int_pred < 1:
            int_pred = 1
        if int_pred > 5:
            int_pred = 5

        df.at[index, 'pred_int_rating'] = round(int_pred)
    return df
```

prediction_service.py

```
import pandas as pd
import os
import glob
import tensorflow as tf
from batch_predictor import predict_batch

# Set working directory
os.chdir('../..')
print(os.getcwd())

# Declare path variables

production_data = 'ml-1k_prod'
prediction_data_dir = f'production/production_data/{production_data}.csv'
PIPELINE_NAME = 'DCN-iterate'
PIPE_DIR = 'pipeline'
SERVING_MODEL_DIR = os.path.join(PIPE_DIR, 'serving_model', PIPELINE_NAME)

# Load prediction dataset and model
df = pd.read_csv(prediction_data_dir)
loaded_model = tf.saved_model.load(max(glob.glob(os.path.join(SERVING_MODEL_DIR, '*/')),
key=os.path.getmtime))  # Load newest model from 'serving_model'
model_name = max(glob.glob(os.path.join(SERVING_MODEL_DIR, '*/')), key=os.path.getmtime).split("/")[-
2]  # Get model name to assign monitoring data to model

MONITORING = f'production/monitoring/{model_name}_monitoring.csv'

# Add rating columns to be filled by predict_batch
df['pred_rating'] = 0
df['pred_int_rating'] = 0

# Fill in columns for float prediction value and int prediction value
df_pred = predict_batch(df, loaded_model)

# Add predictions to a monitoring table (simple CSV file)
if os.path.exists(MONITORING):
    df_monitor = pd.read_csv(MONITORING)
    df_joint = pd.concat([df_monitor, df_pred], ignore_index=True)
    df_joint.to_csv(MONITORING, index=False)
else:
    df_pred.to_csv(MONITORING, index=False)
```

prediction_service.py

```
# Takes in a model and a dataset entry (in form of a dictionary) and returns the predicted value of
the model

def prediction_server(model, entry):
    score = model({'user_id': [[entry['user_id']]], 'movie_id': [[entry['movie_id']]], 'user_gender':
[[entry['user_gender']]], 'user_occupation': [[entry['user_occupation']]], 'user_age_cohort': [[en-
try['user_age_cohort']]]}).numpy()
    return score[0][0][0]
```

References

References

Aggarwal, C. C. (2016). *Recommender Systems: The Textbook*. Cham: Springer International Publishing.

Algorithmia. *2020 state of enterprise machine learning*. Retrieved from https://info.algorithmia.com/hubfs/2019/Whitepapers/The-State-of-Enterprise-ML-2020/Algorithmia_2020_State_of_Enterprise_ML.pdf

Alla, S., & Adari, S. K. (2021). *Beginning MLOps with MLFlow*. Springer.

Alyari, F., & Jafari Navimipour, N. (2018). Recommender systems. *Kybernetes*, *47*(5), 985–1017. https://doi.org/10.1108/K-06-2017-0196

Apache Software Foundation (n.d.). Airflow. Retrieved from https://airflow.apache.org/

Arrikto (n.d.). Arrikto Enterprise Kubeflow Documentation. Retrieved from https://docs.arrikto.com/

Baena-Garcıa, M., Del Campo-Ávila, J., Fidalgo, R., Bifet, A., Gavalda, R., & Morales-Bueno, R. (2006). Early drift detection method. In *Fourth international workshop on knowledge discovery from data streams*.

Baker, T. (2019). *Smarter Humans. Smarter Machines.* Retrieved from Refinitiv website: https://www.refinitiv.com/en/resources/special-report/refinitiv-2019-artificial-intelligence-machine-learning-global-study

Baylor, D., Breck, E., Cheng, H.-T., Fiedel, N., Foo, C. Y., Haque, Z., . . . Zinkevich, M. (2017). TFX: A TensorFlow-Based Production-Scale Machine Learning Platform. In S. Matwin, S. Yu, & F. Farooq (Eds.), *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (pp. 1387–1395). New York, NY, USA: ACM. https://doi.org/10.1145/3097983.3098021

Blondel, M., Fujino, A., Ueda, N., & Ishihata, M. (2016, July 25). *Higher-Order Factorization Machines*. Retrieved from http://arxiv.org/pdf/1607.07195v2

Blondel, M., Ishihata, M., Fujino, A., & Ueda, N. (2016, July 29). *Polynomial Networks and Factorization Machines: New Insights and Efficient Training Algorithms*. Retrieved from http://arxiv.org/pdf/1607.08810v1

Charniak, E. (2019). *Introduction to Deep Learning*. The MIT Press.

Choy, G., Khalilzadeh, O., Michalski, M., Do, S., Samir, A. E., Pianykh, O. S., . . . Dreyer, K. J. (2018). Current Applications and Future Impact of Machine Learning in Radiology. *Radiology*, *288*(2), 318–328. https://doi.org/10.1148/radiol.2018171820

Chui, M., Hall, B., Singla, A., & Sukharevsky, A. (2021, December 8). *The state of AI in 2021*. Retrieved from McKinsey website: https://www.mckinsey.com/business-functions/quantumblack/our-insights/global-survey-the-state-of-ai-in-2021

Columbus, L. (2017, July 9). McKinsey's State Of Machine Learning And AI, 2017. *Forbes*. Retrieved from https://www.forbes.com/sites/louiscolumbus/2017/07/09/mckinseys-state-of-machine-learning-and-ai-2017/?sh=63414b1b75b6

Covington, P., Adams, J., & Sargin, E. (2016). Deep Neural Networks for YouTube Recommendations. In S. Sen, W. Geyer, J. Freyne, & P. Castells (Eds.), *Proceedings of the 10th ACM Conference on Recommender Systems* (pp. 191–198). New York, NY, USA: ACM. https://doi.org/10.1145/2959100.2959190

Crowe, R. (2019). Why do I need metadata? (TensorFlow Extended). Retrieved from https://www.youtube.com/watch?v=cc1-eocgm1E

Cunningham, W. (1993). The WyCash portfolio management system. *ACM SIGPLAN OOPS Messenger*, *4*(2), 29–30. https://doi.org/10.1145/157710.157715

Dacrema, M. F., Boglio, S., Cremonesi, P., & Jannach, D. (2021). A Troubling Analysis of Reproducibility and Progress in Recommender Systems Research. *ACM Transactions on Information Systems*, *39*(2), 1–49. https://doi.org/10.1145/3434185

Denis Baylor, Kevin Haas, Konstantinos Katsiapis, Sammy Leong, Rose Liu, Clemens Menwald, . . . Martin Zinkevich (2019). Continuous Training for Production ML in the TensorFlow Extended (TFX) Platform. In *2019 USENIX Conference on Operational Machine Learning (OpML 19)* (pp. 51–53). Santa Clara, CA: USENIX Association. Retrieved from https://www.usenix.org/conference/opml19/presentation/baylor

Deo, N. (2019). *Graph theory: With applications to engineering & computer science*. *Dover books on mathematics*. Mineola, NY: Dover Publications.

Feng Gu, Zhang, G., Jie Lu, & Chin-Teng Lin (2016). Concept drift detection based on equal density estimation. In *2016 International Joint Conference on Neural Networks (IJCNN)*.

Fernández-Tobias, I., Cantador, I., Kaminskas, M., & Ricci, F. (2012). Cross-domain recommender systems: A survey of the state of the art. In *Spanish conference on information retrieval.* Symposium conducted at the meeting of sn.

Fleder, D. M., & Hosanagar, K. (2007). Recommender systems and their impact on sales diversity. In J. MacKie-Mason, D. Parkes, & P. Resnick (Eds.), *Proceedings of the 8th ACM conference on Electronic commerce - EC '07* (p. 192). New York, New York, USA: ACM Press. https://doi.org/10.1145/1250910.1250939

Fowler, M. (2019). *Refactoring: Improving the design of existing code* (Second edition). *Addison-Wesley signature series*. Boston: Addison-Wesley.

Gama, J. [João], Medas, P., Castillo, G., & Rodrigues, P. (2004). Learning with Drift Detection. In D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C.

Mitchell, . . . S. Labidi (Eds.), *Lecture Notes in Computer Science. Advances in Artificial Intelligence – SBIA 2004* (Vol. 3171, pp. 286–295). Berlin, Heidelberg: Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-28645-5_29

Gama, J. [João], Žliobaitė, I., Bifet, A., Pechenizkiy, M., & Bouchachia, A. (2014). A survey on concept drift adaptation. *ACM Computing Surveys*, *46*(4), 1–37. https://doi.org/10.1145/2523813

Garousi, V., Petersen, K., & Ozkan, B. (2016). Challenges and best practices in industry-academia collaborations in software engineering: A systematic literature review. *Information and Software Technology*, *79*, 106–127. https://doi.org/10.1016/j.infsof.2016.07.006

GOODFELLOW, I., BENGIO, Y., & COURVILLE, A. (2016). *Deep learning*. *Adaptive computation and machine learning series*. Cambridge, Massachusetts, London: MIT Press.

Google LLC (2019a). TFX: TensorFlow Extended (TFX) is an end-to-end platform for deploying production ML pipelines. Retrieved from https://www.tensorflow.org/tfx

Google LLC (2019b). The TFX User Guide. Retrieved from https://www.tensorflow.org/tfx/guide

Google LLC (2020a). MLOps: Continuous delivery and automation pipelines in machine learning.

Google LLC (2020b). Retrieval. Retrieved from https://developers.google.com/machine-learning/recommendation/dnn/retrieval

Google LLC (2020c, February 11). Scoring. Retrieved from https://developers.google.com/machine-learning/recommendation/dnn/scoring

Google LLC (2021a). Collaborative Filtering. Retrieved from https://developers.google.com/machine-learning/recommendation/collaborative/basics

Google LLC (2021b). ExampleValidator. Retrieved from https://www.tensorflow.org/tfx/guide/exampleval

Google LLC (2022). Using TensorFlow Recommenders with TFX. Retrieved from https://www.tensorflow.org/recommenders/examples/ranking_tfx

GroupLens (n.d.). Index of /datasets/movielens. Retrieved from https://files.grouplens.org/datasets/movielens/

Gurney, K. (2014). *An Introduction to Neural Networks*. Hoboken: CRC Press.

Hanin, B. (2019). Universal Function Approximation by Deep Neural Nets with Bounded Width and ReLU Activations. *Mathematics*, *7*(10), 992. https://doi.org/10.3390/math7100992

Harenslak, B., & Ruiter, J. de (2021). *Data Pipelines with Apache Airflow* (1st edition). Manning Publications.

Harper, F. M., & Konstan, J. A. (2016). The MovieLens Datasets. *ACM Transactions on Interactive Intelligent Systems*, *5*(4), 1–19. https://doi.org/10.1145/2827872

Heaton, J. (2012). *Introduction to the Math of Neural Networks*. Heaton Research.

Hevner, March, Park, & Ram (2004). Design Science in Information Systems Research. *MIS Quarterly*, *28*(1), 75. https://doi.org/10.2307/25148625

Hevner, A. (2007). A Three Cycle View of Design Science Research. *Scandinavian Journal of Information Systems*, *19*.

Hevner, A., & Chatterjee, S. (Eds.) (2010). *Integrated Series in Information Systems*. *Design Research in Information Systems*. Boston, MA: Springer US. https://doi.org/10.1007/978-1-4419-5653-8

Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. R. (2012, July 3). *Improving neural networks by preventing co-adaptation of feature detectors*. Retrieved from http://arxiv.org/pdf/1207.0580v1

Huang, T., She, Q., Wang, Z., & Zhang, J. (2020, July 6). *GateNet: Gating-Enhanced Deep Network for Click-Through Rate Prediction*. Retrieved from http://arxiv.org/pdf/2007.03519v1

Hutter, F., Kotthoff, L., & Vanschoren, J. (2019). *Automated Machine Learning*. Springer Nature. Retrieved from https://doi.org/10.1007/978-3-030-05318-5

IMDb (2022). IMDb frontpage.

International Air Transport Association (2022). *The impact of the war in Ukraine on the aviation industry*. Retrieved from https://www.iata.org/en/iata-repository/publications/economic-reports/the-impact-of-the-conflict-between-russia-and-ukraine-on-aviation/

Jannach, D., & Zanker, M. (2022). Value and Impact of Recommender Systems. In F. Ricci, L. Rokach, & B. Shapira (Eds.), *Recommender Systems Handbook* (pp. 519–546). New York, NY: Springer US. https://doi.org/10.1007/978-1-0716-2197-4_14

Jordan, M. I., & Mitchell, T. M. (2015). Machine learning: Trends, perspectives, and prospects. *Science (New York, N.Y.)*, *349*(6245), 255–260. https://doi.org/10.1126/science.aaa8415

Kakantousis, T., Kouzoupis, A., Buso, F., Berthou, G., Dowling, J., & Haridi, S. (2019). Horizontally scalable ml pipelines with a feature store. In *Proceedings of the 2nd SysML Conference, Palo Alto, CA, USA*.

Karmaker, S. K., Hassan, M. M., Smith, M. J., Xu, L., Zhai, C., & Veeramachaneni, K. (2022). AutoML to Date and Beyond: Challenges and Opportunities. *ACM Computing Surveys*, *54*(8), 1–36. https://doi.org/10.1145/3470918

Khusro, S., Ali, Z., & Ullah, I. (2016). Recommender Systems: Issues, Challenges, and Research Opportunities. In K. J. Kim & N. Joukov (Eds.), *Lecture Notes in Electrical Engineering. Information Science and Applications (ICISA) 2016* (Vol. 376,

pp. 1179–1189). Singapore: Springer Singapore. https://doi.org/10.1007/978-981-10-0557-2_112

Knotek, J., & Pereira, W. (2011). Survey on Concept Drift. Retrieved from https://is.muni.cz/el/1433/podzim2011/PA164/um/drift_detection_methods.pdf

Koren, Y. (2009a). The bellkor solution to the netflix grand prize. *Netflix Prize Documentation*, *81*(2009), 1–10.

Koren, Y. (2009b). Collaborative Filtering with Temporal Dynamics. In *KDD '09, Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (pp. 447–456). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/1557019.1557072

Koren, Y., Bell, R., & Volinsky, C. (2009). Matrix factorization techniques for recommender systems. *Computer*, *42*(8), 30–37.

Koren, Yehuda and Rendle, Steffen and Bell, Robert (2022). Advances in Collaborative Filtering. In Ricci, Francesco and Rokach, Lior and Shapira, Bracha (Ed.), *Recommender Systems Handbook* (pp. 91–142). New York, NY: Springer US. https://doi.org/10.1007/978-1-0716-2197-4_3

Koychev, I. (2004). Gradual Forgetting for Adaptation to Concept Drift. *ECAI*.

Kreuzberger, D., Kühl, N., & Hirschl, S. (2022, May 4). *Machine Learning Operations (MLOps): Overview, Definition, and Architecture*. Retrieved from http://arxiv.org/pdf/2205.02302v3

Kubeflow (n.d.). Kubeflow. Retrieved from https://www.kubeflow.org/

LeCun, Y., Boser, B., Denker, J., Henderson, D., Howard, R., Hubbard, W., & Jackel, L. (1989). Handwritten Digit Recognition with a Back-Propagation Network. In D. Touretzky (Ed.), *Advances in Neural Information Processing Systems* (Vol. 2). Morgan-Kaufmann. Retrieved from https://proceedings.neurips.cc/paper/1989/file/53c3bce66e43be4f209556518c2fcb54-Paper.pdf

Lian, J., Zhou, X., Zhang, F., Chen, Z., Xie, X., & Sun, G. (2018). xDeepFM. In Y. Guo & F. Farooq (Eds.), *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (pp. 1754–1763). New York, NY, USA: ACM. https://doi.org/10.1145/3219819.3220023

Liu, A., Zhang, G., & Lu, J. (2017). Fuzzy time windowing for gradual concept drift adaptation. In *2017 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE).* Symposium conducted at the meeting of IEEE.

Lo, Y.-Y., Liao, W., Chang, C.-S., & Lee, Y.-C. (2018). Temporal Matrix Factorization for Tracking Concept Drift in Individual User Preferences. *IEEE Transactions on Computational Social Systems*, *5*(1), 156–168. https://doi.org/10.1109/TCSS.2017.2772295

Loehlin, J. C., & Beaujean, A. A. (2017). *Latent variable models: An introduction to factor, path, and structural equation analysis* (5. ed.). New York: Routledge.

López-Sánchez, D., Herrero, J. R., Arrieta, A. G., & Corchado, J. M. (2018). Hybridizing metric learning and case-based reasoning for adaptable clickbait detection. *Applied Intelligence*, *48*(9), 2967–2982. https://doi.org/10.1007/s10489-017-1109-7

Lu, J., Liu, A., Dong, F., Gu, F., Gama, J. [Joao], & Zhang, G. (2018). Learning under Concept Drift: A Review. *IEEE Transactions on Knowledge and Data Engineering*, 1. https://doi.org/10.1109/TKDE.2018.2876857

Makinen, S., Skogstrom, H., Laaksonen, E., & Mikkonen, T. (2021). Who Needs MLOps: What Data Scientists Seek to Accomplish and How Can MLOps Help? In *2021 IEEE/ACM 1st Workshop on AI Engineering - Software Engineering for AI (WAIN)* (pp. 109–112). IEEE. https://doi.org/10.1109/WAIN52551.2021.00024

May, R. J., Maier, H. R., & Dandy, G. C. (2010). Data splitting for artificial neural networks using SOM-based stratified sampling. *Neural Networks : The Official Journal of the International Neural Network Society*, *23*(2), 283–294. https://doi.org/10.1016/j.neunet.2009.11.009

Minsky, M. (1961). Steps toward Artificial Intelligence. *Proceedings of the IRE*, *49*(1), 8–30. https://doi.org/10.1109/JRPROC.1961.287775

Miranda, L. J. (2021). Towards data-centric machine learning: a short review. *Ljvmiranda921. Github. Io*.

Nair, V., & Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. In *Icml*.

Ng, A., Crowe, R., & Moroney, L. (2021). Introduction to Machine Learning in Production. Retrieved from https://de.coursera.org/learn/introduction-to-machine-learning-in-production?specialization=machine-learning-engineering-for-production-mlops

Pandas (n.d.). pandas. Retrieved from https://pandas.pydata.org/

Qahtan, A. A., Alharbi, B., Wang, S., & Zhang, X. [Xiangliang] (2015). A PCA-Based Change Detection Framework for Multidimensional Data Streams: Change Detection in Multidimensional Data Streams. In *KDD '15, Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (pp. 935–944). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/2783258.2783359

Refinitiv (2020). *THE RISE OF THE DATA SCIENTIST:: Machine learning models for the future*. Retrieved from https://www.refinitiv.com/en/resources/special-report/refinitiv-2020-artificial-intelligence-machine-learning-global-study

Rendle, S., Krichene, W., Zhang, L., & Anderson, J. (2020). Neural Collaborative Filtering vs. Matrix Factorization Revisited. In *Fourteenth ACM Conference on Recommender Systems* (pp. 240–248). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/3383313.3412488

Renggli, C., Rimanic, L., Gürel, N. M., Karlaš, B., Wu, W., & Zhang, C. (2021, February 15). *A Data Quality-Driven View of MLOps*. Retrieved from http://arxiv.org/pdf/2102.07750v1

Rimol, M. (2021, November 22). *Gartner Forecasts Worldwide Artificial Intelligence Software Market to Reach $62 Billion in 2022*. Retrieved from Gartner website: https://www.gartner.com/en/newsroom/press-releases/2021-11-22-gartner-forecasts-worldwide-artificial-intelligence-software-market-to-reach-62-billion-in-2022

Rosenblatt, F. (1961). *Principles of neurodynamics. perceptrons and the theory of brain mechanisms.*

Ruf, P., Madan, M., Reich, C., & Ould-Abdeslam, D. (2021). Demystifying MLOps and Presenting a Recipe for the Selection of Open-Source Tools. *Applied Sciences*, *11*(19), 8861. https://doi.org/10.3390/app11198861

Salama, K., Kazmierczak, J., & Schut, D. (2021). Practitioners guide to MLOps: A framework for continuous delivery and automation of machine learning. Retrieved from https://services.google.com/fh/files/misc/practitioners_guide_to_mlops_whitepaper.pdf

Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., . . . Dennison, D. (2015). Hidden technical debt in machine learning systems. *Advances in Neural Information Processing Systems*, *28*(2), 2503–2511.

Serban, A., van der Blom, K., Hoos, H., & Visser, J. (2020). Adoption and Effects of Software Engineering Best Practices in Machine Learning, *1*, 1–12. https://doi.org/10.1145/3382494.3410681

Shan, Y., Hoens, T. R., Jiao, J., Wang, H., Yu, D., & Mao, J. C. (2016). Deep Crossing. In B. Krishnapuram, M. Shah, A. Smola, C. Aggarwal, D. Shen, & R. Rastogi (Eds.), *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (pp. 255–262). New York, NY, USA: ACM. https://doi.org/10.1145/2939672.2939704

Singh, P. K., Choudhury, P., Dey, A. K., & Pramanik, P. K. D. (2021). Recommender systems: an overview, research trends, and future directions. *International Journal of Business and Systems Research*, *15*(1), 14. https://doi.org/10.1504/ijbsr.2021.10033303

Studer, S., Bui, T. B., Drescher, C., Hanuschkin, A., Winkler, L., Peters, S., & Müller, K.-R. (2021). Towards CRISP-ML(Q): A Machine Learning Process Model with Quality Assurance Methodology. *Machine Learning and Knowledge Extraction*, *3*(2), 392–413. https://doi.org/10.3390/make3020020

Su, X., & Khoshgoftaar, T. M. (2009). A Survey of Collaborative Filtering Techniques. *Advances in Artificial Intelligence*, *2009*, 1–19. https://doi.org/10.1155/2009/421425

Symeonidis, G., Nerantzis, E., Kazakis, A., & Papakostas, G. A. (2022). MLOps - Definitions, Tools and Challenges. In *2022 IEEE 12th Annual Computing and Communication Workshop and Conference (CCWC)* (pp. 453–460). IEEE. https://doi.org/10.1109/CCWC54503.2022.9720902

Tamburri, D. A. (2020). Sustainable MLOps: Trends and Challenges. In *2020 22nd International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)* (pp. 17–23). IEEE. https://doi.org/10.1109/SYNASC51798.2020.00015

TensorFlow (n.d.). TensorFlow Recommenders. Retrieved from https://www.tensorflow.org/recommenders

TensorFlow (2022a). Deep & Cross Network (DCN). Retrieved from https://www.tensorflow.org/recommenders/examples/dcn

TensorFlow (2022b). TFRS API: All symbols in TensorFlow Recommenders. Retrieved from https://www.tensorflow.org/recommenders/api_docs/python/tfrs/all_symbols

Theodoridis, J., & Grießhaber, D. (n.d.). Deeplearning Cluster - Docs. Retrieved from https://deeplearn.pages.mi.hdm-stuttgart.de/docs/

Vellido, A., Lisboa, P. J., & Meehan, K. (2000). Quantitative Characterization and Prediction of On-Line Purchasing Behavior: A Latent Variable Approach. *International Journal of Electronic Commerce*, *4*(4), 83–104. https://doi.org/10.1080/10864415.2000.11518380

Wang, R., Fu, B., Fu, G., & Wang, M. (2017, August 17). *Deep & Cross Network for Ad Click Predictions*. Retrieved from http://arxiv.org/pdf/1708.05123v1 https://doi.org/AdKDD

Wang, R., Shivanna, R., Cheng, D., Jain, S., Lin, D., Hong, L., & Chi, E. (2021). DCN V2: Improved Deep & Cross Network and Practical Lessons for Web-scale Learning to Rank Systems. In J. Leskovec, M. Grobelnik, M. Najork, J. Tang, & L. Zia (Eds.), *Proceedings of the Web Conference 2021* (pp. 1785–1797). New York, NY, USA: ACM. https://doi.org/10.1145/3442381.3450078

Yan, Y., & Li, L. (2020). xDeepInt: a hybrid architecture for modeling the vector-wise and bit-wise feature interactions.

Yavuz, B., & Chockalingam, P. (2019). Introducing Delta Time Travel for Large Scale Data Lakes. Retrieved from https://databricks.com/de/blog/2019/02/04/introducing-delta-time-travel-for-large-scale-data-lakes.html