Mats Eikeland Mollestad

Mitigating Hidden Technical Debt in Machine Learning Systems

master project, spring 2023

Artificial Intelligence Group
Department of Computer and Information Science
Faculty of Information Technology, Mathematics and Electrical Engineering

Abstract

The paper Sculley et al. [2015] is recognized as one of the most influential works within the Machine Learning Operations community. The technical debts identified therein have led to significant improvements in the deployment of ML systems, particularly concerning artifact version control, reproducibility, and infrastructure. However, existing tools have predominantly focused on the model and the associated software, leaving data related debts largely unaddressed, as evidenced by conducted interviews.

This project introduces a novel method specifically targeting data-related debt. This approach results in improved ML products and reduces the time spent understanding the system's data flow. This is achieved by offering a metadata compiler that automatically captures data dependencies, describes data schemas, standardise feature engineering, and simplifies the understanding of model logic. The collection of dependency metadata would otherwise be too inconvenient to manually define. The obtained metadata can then be empowered by an implementation compiler that optimize the system and decrease development costs by generating common components essential for Machine Learning Operations (MLOps).

Preface

I would like to thank my supervisor Anders Kofod-Petersen for making this master thesis possible, and providing feedback. Furthermore, I would like to thank everyone that took the time to be interview about their problems when deploying ML, but also for testing the proposed solution.

Lastly, I would like to thank Otovo ASA for allowing me to write a master thesis, while working part time on practical problems related to the master thesis. Therefore, making it clearer why such research would be useful.

Mats Eikeland Mollestad
Trondheim, June 13, 2023

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent years, machine learning (ML) has become an integral part of many industries, transforming them in unprecedented ways. However, this transformation does not come without challenges. One of the most prominent issues is the accumulation of 'technical debt', a term used to describe the future costs incurred by short-term decisions made during the development and deployment of ML systems. These costs often manifest as increased complexity and a lack of system-level visibility.

While technical debt in ML systems has been a topic of research, most existing solutions are predominantly focused on isolated model components. This approach overlooks the systemic nature of the problem and fails to provide a comprehensive solution. Furthermore, the split between ML research and the realities of deploying ML systems in practice can exacerbate this debt, leading to components that don't integrate well into an ML system and demanding extensive resources.

Given these issues, this study aims to go beyond the model-centric focus of current approaches. It seeks to examine the types of technical data debt common to ML systems and develop a data-centric approach that could manage data flow at a system-wide level. This approach, which involves leveraging metadata, has the potential to reduce technical debt and enhance the quality of ML systems.

The evaluation of this data-centric method will be grounded in the types of technical debt as described by Sculley et al. [2015], offering a robust framework to assess the effectiveness of this novel approach.

1.1 Background and Motivation

This research is fueled by the critical need to tackle the increasing technical debt intrinsic to ML systems. Elements such as limited system visibility, surplus glue code, dead code, and data dependency debt can lead to rapid degradation of ML systems. Semi-structured interviews with industry experts corroborate the urgency of addressing these problems. It is proposed that categorizing ML models into three core components - the model ($f(X)$), the input (X), and the output (y) - could provide a simplified framework for data management. Enabling a more efficient strategy for managing technical debt in ML systems, as logical and functional can be generated, similar to how software 2.0 works as illustrated in figure (2.2).

1.2 Goals and Research Questions

This study aims to enhance understanding of the unique data challenges involved in deploying ML systems and how to address them.

Research question 1 *What specific types of technical data debt commonly occur in ML systems?*

Research question 2 *What methods can be developed to increase visibility of data debt in ML systems?*

Research question 3 *How can metadata be leveraged to mitigate technical debt and improve ML system quality?*

1.3 Research Method

This study adopts a mixed-methods research approach to explore hidden technical debt in machine learning systems. Our methodology involves an analysis of Sculley et al. [2015], a key paper on technical debt in machine learning, which serves as the benchmark for evaluating our proposed solution.

In addition, the research draws on insights from reputable industry blogs and authoritative books for broader context and practical perspectives. Moreover, we conducted interviews with field experts, complementing the theoretical basis with real-world experiences and insights.

The proposed solution's efficacy is assessed through a systematic comparison against the technical debts outlined in Sculley et al. [2015]. While also conducting interviews comparing the proposed solution against other existing solutions.

1.4 Contributions

This research seeks to address a knowledge gap within data management for ML systems by developing and releasing new tools aimed at mitigating the identified challenges. The findings will subsequently be shared with the MLOps Community in Oslo in May.

1.5 Thesis Structure

The thesis commences with a discussion of the theoretical background, including the differences between ML research and deployment, common ML life-cycles, and an exploration of various ML components. Following this, we share insights from our interviews and present a detailed breakdown of technical debt as defined by Sculley et al. [2015]. Building on this understanding, we introduce a simplified representation of ML systems that forms the theoretical basis for our proposed solution, Aligned. The final chapters evaluate the efficacy of Aligned in mitigating the technical debt identified earlier in the thesis.

Chapter 2

Background Theory and Motivation

In this chapter, we will present the background theory found in the literature and the concepts discussed in the interviews.

2.1 Background Theory

Here will theory related to the master thesis be presented. The theory works as the foundation for the proposed solution in chapter (3). It will cover a combination of data science, data engineering, and ML related topics needed to understand the needs of a deployed ML system.

2.1.1 Software 2.0

The traditional method when creating software involves detecting a problem, designing a system to solve it, and writing code to enforce that system, as shown in figure (2.1). This approach has been highly effective and has resulted in significant value in many domains. However, selecting the right system and rules can be difficult, as it often relies on subjective knowledge. This makes it difficult to know if other models may perform better. This approach also struggles with complex tasks like image detection, as it is hard to create the correct rules. Classical computer vision tasks use this approach with methods like HoG for object detection William T. Freeman [1994].

As a result, a new paradigm has emerged that changes the approach. Instead of defining rules in code, we define a way of finding the best pattern based on a set of scenarios. Allowing us to find high-quality solutions even in complex

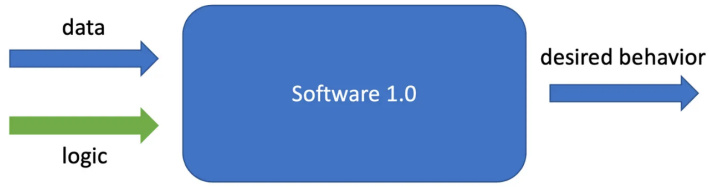

Figure 2.1: Software 1.0

situations where people struggle to find a clear rule set, as shown in figure (2.2). One such method would be machine learning where we only define the intended behavior.

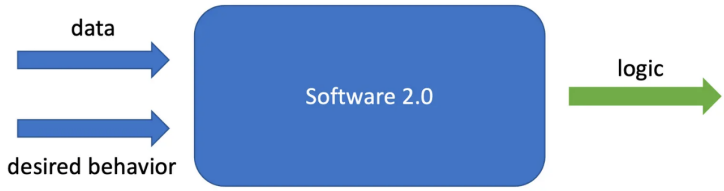

Figure 2.2: Software 2.0

2.1.2 Types of ML

Before we explore the requirements for deploying ML and the challenges that arise, it is important to understand the different types of ML and how they behave. Knowing the distinctions between different types of ML can help us better understand their capabilities and limitations, and inform our approach to deploying them in real-world scenarios. Before we go to deep into what is needed to deploy ML, and the problems that arise, will it be important to know about the different types of ML that exists, and how they behave.

Supervised

Maybe the most common type of ML, where the problem statement is find a function $f(x) = y$. Therefore making it possible to use the function in order to predict $\hat{y}$ in scenarios where y is unknown.

Here could the output y either be a scalar for regression tasks, or a categorical value for classification tasks. And some examples of models that perform such a task are logistic regression Hosmer Jr et al. [2013], decision trees Drucker and Cortes [1995], boosted trees Drucker and Cortes [1995], and neural nets Gardner and Dorling [1998].

Unsupervised

Furthermore, do we have unsupervised where we have unlabeled data. Resulting in a function that structure the data by it self. Some examples of this would be K-means clustering Likas et al. [2003], DBSCAN Schubert et al. [2017], but it can also be an auto-encoder which extracts features Meng et al. [2017].

Reinforcement

Lastly reinforcement learning which tries to find a function that maximises a reward function within an environment Li [2017]. This is therefore used to model evolution, where the best species gets to live on, for each generation. Such models can also be a neural net Gardner and Dorling [1998].

2.1.3 ML research vs. deployment

Before we go into the theory and concepts needed to deploy machine learning, it is important to understand the different objectives of ML -research and -deployment. While there are similarities between these two objectives, there are also important differences.

Goal and problem statement

Machine learning researchers goal is to improve existing knowledge or discover new knowledge. However, according to Ng [2021], roughly 90% of research uses model-centric approaches. This means that researchers often alter the architecture of their models in order to improve performance. Therefore, leading to standardized data sets and standardized evaluation metrics.

However, this approach can be problematic. Standardized and cleaned data sets do not always reflect real-world data, which means that model-centric research may not be applicable in some cases.

When it comes to deploying ML, the goals and the constraints change quite a lot. The objective of a ML product is not to discover new knowledge. It is to create business value.

This leads to a slight change in the problem statement. Instead of trying to have state-of-the-art model evaluation metrics, it is more important to understand which metrics align with the business goal and which solutions fits the cost constraints and development time.

Constraints

For instance, it will not make sense for most businesses to train a large model for months just to get a minor increase in accuracy. Therefore, loss of valuable time and money. It might make more business sense to deploy a slightly worse model because it is faster and cost less. Therefore, increasing the return on investment.

Data sets

One core fundamental difference is how the data sets behave. As stated in section (2.1.3), research uses standardized data sets. These are often well-structured data sets with high label and feature quality. Resulting in a focus on the model. However, the main challenge in a business setting is often related to data quality and finding the ground truth label. Therefore, leading to few changes in the model. As a result, machine learning in businesses often relies more upon improving data quality and not the algorithm itself. This is somewhat mentioned in all ML lectures in the saying *garbage in, garbage out*. However, research has started embracing this even more, and started focusing on fields like active learning, and semi-supervised learning, which will be discussed 2.1.14. Therefore, moving away from the model-centric focus, and focusing more on data-centric which is more aligned with business goals.

Risk

Furthermore, there will be a need for more risk analytics. Reflecting on how the effect of incorrect predictions will have can be one risk to evaluate. Using Zillow[1] as an example were they lost 420 million dollars from their house-flipping model partially because they got the evaluation process incorrect Times [2022]. They had an average accuracy of around 96-99%, which sounds like a high-quality model. But this is misleading, as it misses one of the most crucial parts, the errors. Because the model can still predict the bid and sell price correctly 99% of the time. However, one overpriced prediction can ruin everything, as the loss can outway the gain. Therefore, it is not about how many times the model was

[1] Zillow - `https://www.zillow.com`

right but rather how many times the model was incorrect, as this is the scenario that is the most damaging for the business Tenenholtz [2022].

Trust

Lastly, explainability can be more valuable then the predictions them self. Especially in human critical applications where people can get hurt. Something like Tesla's auto-pilot model where one error can be life-threatening. Therefore, people would struggle to trust the system if it was a black box, leading to no users and zero value gain. This can result in using a worse performing model but an increase in value because of increased trust.

2.1.4 MLOps

In the past years have the field of Machine Learning Operation (MLOps) emerged, and started getting some traction, but what is it actually?

MLOps is described as the principles, used to bridge the gap between development and operating machine learning products Kreuzberger et al. [2022]. Therefore, MLOps will be essential for a business to measure value gain, and maintain it effectively over time. However this covers a wide range of topics, both technical, and cultural. And the cultural part is usually the hardest part to get right.

The reason being that many people need to be convinced that the extra problems that MLOps is trying to solve, actually are problems in the first place. When diving a bit more into it, will it become a bit clearer why, but in short is it because a lot of the problems can become silent killers. Therefore making it hard to detect the problems unless someone else have experienced them before.

ML Engineer role

The rise of MLOps has led to the emergence of a new role: the *ML Engineer*. This role requires a combination of skills typically associated with five different roles: Data Scientist, Software Engineer, Data Engineer, DevOps Engineer, and Backend Engineer Kreuzberger et al. [2022].

Given the extensive skillset required, it is challenging to find personnel who are qualified for the role. For many smaller businesses, it may be cost-prohibitive to hire adequately experienced individuals. However, this challenge can lead to using untrained personnel in a unfamiliar domain. Such as using a software engineer as a data engineer, a data engineer as a data scientist, and a data scientist as a software engineer. Therefore, potentially using bad practices, which increases technical debt and reduces flexibility.

2.1.5 Model development life-cycle

Before delving into the pain-points of developing ML models, it's crucial to understand the life-cycle of a model.

There are various diagrams aiming to present the life-cycle of a ML product. However, they generally follow the structure described below.

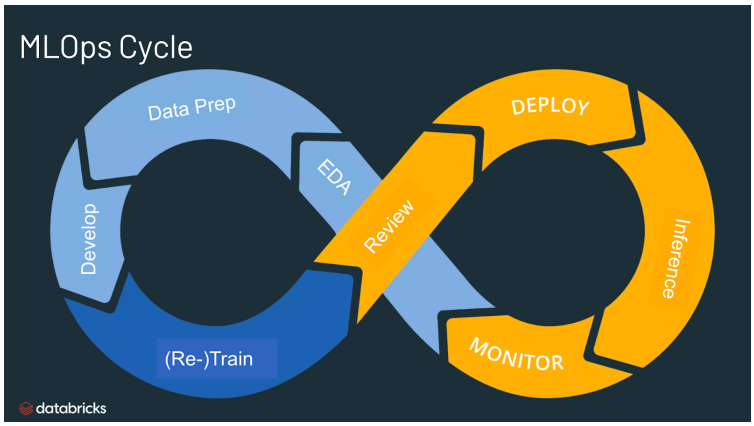

Figure 2.3: MLOps life-cycle Databricks [2022]

Scoping and problem statement

One of the most crucial stages of the life-cycle is scoping and problem statement. This stage involves defining the problem and potential value gain. It requires identifying existing pain points and formulating them into a problem that ML can help resolve. This involves questioning its feasibility and the available data make it possible. Lastly, and arguably the hardest part, is to figure out how to measure the model's impact concerning the business goals.

Companies like Spotify[2] emphasize that ML products should never be the default method, but rather a tool used to reduce the friction for existing solutions. Therefore, they always have a manual solution before an ML product is introduced Budelman [2019].

[2]Spotify - `https://open.spotify.com`

Data collection

After the scoping and problem statement have been defined, the next step is to collect the appropriate data to facilitate the project.

The ease of this process can vary depending on the business's maturity. Initially, the most important aspect is knowing what data exists and where it can be accessed. Ideally, you'd like to browse through a data catalog, as you might through a shop until you find what you want. However, this requires extensive documentation of all data schemas and infrastructure. Perhaps the data exists in an OLTP SQL, or NoSQL database. Maybe it resides in an OLAP data warehouse or even a data lake. The underlying data storage, in the worst case, can halt an entire ML project if set up incorrectly Kleppmann [2017] Huyen [2022].

Furthermore, the data documentation might be non-existent or outdated, making it difficult to be certain that the used data is as expected.

Nonetheless, if the data is well-documented, other issues may arise. A silent yet dangerous problem is *leaky data*, one type being *point-in-time incorrect data*, which will be discussed more in section (2.1.8) Huyen [2022]. Additionally, knowing what constitutes valid data points, which to exclude, can be challenging. Lastly, the data can behave differently from initial expectations.

Data processing

The next step involves pre-processing and analyzing the data. The need for pre-processing depends on the ML problem at hand.

For instance, an image task may require normalizing the image, reducing noise, or scaling the image. However, an NLP task might compute BoW, TF-IDF (Baeza-Yates et al. [1999]), word vectors (Maas et al. [2011]), or perhaps extract something like PoS values. A tabular problem, on the other hand, may compute ratios between features, PCA, standard scaling, or encode categorical features.

This is also where all the data cleaning would take place. Something like filling in missing values, or fixing typos. Another useful technique would be to add data quality check tools. Therefore, we would be notified if we have an assumption about the data that do not hold. One such assumption could be that we will always have a value between 0 and 1. Tooling enabling this would be Great Expectations[3], or Pandera[4].

[3]Great Expectations - `https://greatexpectations.io`
[4]Pandera - `https://pandera.readthedocs.io/en/stable/index.html`

Model creation

Finally, the task mostly considered the ML part is modeling. This is where the model is architected, constructed, and trained. Again, the model used will depend heavily on which problem is being solved. However, the transformer (Vaswani et al. [2017]) has been shown to perform well for text and image tasks. For tabular data sets, boosting trees (Drucker and Cortes [1995]) are among the top-performing models.

However, as Spotify and many well-known people argue, it's always better to start with a simple baseline model, no matter what. There are many reasons for this, like making it clearer that the advanced models are actually worth the time-investments when coming to the evaluation section (2.1.5). It may also be a good idea to give the model purposely an unfair advantage and see how well it can perform, to set an absolute max performance metric as well. However, there are also other advantages like making sure the data actually have a detectable signal at all, which can be found by purposefully overfitting on a smaller data-set.

Model evaluation

This is another make-or-break step in the ML life-cycle. The evaluation step is where we finally figure out how good our model perform and whether it is good enough to provide business value. However, evaluating a model is multi-layered, and often a deployed model will require more testing than what a research model needs.

The most basic testing process is as follows. Use a held out test set, like a validation test set, and compute performance metrics. The type of metric will depend on which type of problem we solve, as described in section (2.1.2). However, when looking at a classification task, evaluation metrics often include precision, recall, accuracy, F1 score, ROC, AUC, and confusion matrixes.

However, while getting a good accuracy or F1 score can be suitable in an academic setting, extra more specific metrics are often needed to provide business value.

For instance, there is really no value in an ML model if it can be outperformed by a simple, low-cost solution Huyen [2022]. It is therefore important to compare against earlier and simpler models. Comparing against other models also makes it possible to setup new evaluation metrics within a business setting. Something like increased revenue gain, which can lead to setting up estimated ROI metrics.

Doing such comparisons will be aggregated from the other base metrics, but this will make it easier to see if more simplistic models make more sense within a business setting, adding extra steps to validate that we are not pursuing flashy tech trends.

However, just setting up good metrics to track and evaluate is half of the

problem. If the evaluation metrics are evaluated on a poor dataset, the metrics will be useless or, even worse, misleading. Just imagine trying to evaluate a stock price model that evaluates on stock prices before COVID, also known as a bull market[5]. But such an evaluation method would have lead to misleading metrics and lead to unrealistic confidence when applying the model on the stock market in 2022, as the SP 500 decreased in value. Signifying a bear market and leading to concept drift discussed in section (2.1.7).

This problem has been raised, and solutions have been proposed.

First up, the integration test. This is when we test on the freshest data available. So if a model is trained on a year worth of data, but it has one week left of data that is not used. Then the integration test would be that last week of data. Making the test reflect how well a model is performing on the current state of the system. Here will the data often come through a data-stream.

Next, *regression test*. Inspired by regression test within classical software development Sommerville [2004]. This is when we have a separate data set, that contains all the errors a model has predicted previously. Such a data-set will be useful to check that a model is learning from it's mistakes, and that we do not introduce errors that previously was fixed. For instance if we have learned a self-driving model that yellow lights means, "you need to stop soon", but then an update to the model forgets this. That is an unwanted situation, and a regression test set can make it possible to catch such errors earlier.

Minimal functional tests. Somewhat related to what we would call a unit test in software development Ribeiro et al. [2020]. This is when we test a very simple function, and it is expected that this will succeed. Basically, if it fails at the following tests, will the model be failing at it's core functionality. Examples of this within a NLP would be that the model detects a negation PoS in the following sentence "I don't like to fly". However, if we remove the "don't" should the model have not negation PoS in the result.

A slight variation is an *invariance test*. This is when we have a data-set that contains data that should not change the output of a model Ribeiro et al. [2020]. This will therefore test that the model is resilience against unrelevant data patterns. Examples of such tests could be in a facial detection application. let's say we shown two images to the model, but the only difference is that the person has changed ethnicity. This should not effect the model in any way, it should still return the same result as before.

Lastly another variation, *directional test*. Instead of saying that the data should not be effected by the change in input, do directional tests test that the model changes Ribeiro et al. [2020]. One example here could be a house pricing model. If the only thing we changed to the input is that the area of the house is

[5] `https://www.cnbc.com/2021/08/16/sp-500-doubles-from-its-pandemic-bottom-marking-the-fastest-bull-ma`
`html`

double then the baseline, the the model should increase the price. We will either make us catch models that do not follow the expected pattern, or we will find that the problem we try to solve actually do not follow our predefined beliefs.

Model deployment

Finally, we can deploy a model that provides some value. Or at least this is what we think based on our previous steps. Even though we have set up thorough testing can we not be completely sure it works as expected. And deploying a bad model can be costly. This is why different deployment strategies have been developed.

The deployment strategies will be inspired by classical software. However with a slightly more scientific spin.

Firstly, *shadowing*. One very common practice when deploying a new model version is to set it up as a shadow model. This is when a model is running in a sort of *stealth mode*, therefore not effecting the users experience, but only generating prediction for later usage, and analysis. As LinkedIn[6] have noticed in a blog-post, is this often their first go to method when deploying a new model. Therefore making sure that the new model do not negatively effect the business, in case the model is poor.

Secondly, *A/B testing*. This is when you randomly distribute the requests to the different models equally, and then compare the performance at a later stage to see which worked best Sommerville [2004].

Canary deployment is a commonly used within DevOps tooling to make sure that the new solution can support all the requests. Canary models will therefore receive a small fraction of the requests in the beginning, and slowly get more and more if everything works out fine Huyen [2022].

Finally, did I want to mention *bandits* which is a lesser known strategy for deploying new models. The reasoning behind it is that we can use statistics to provide us with a better understanding and choose the model that is most likely to perform the best Dutta et al. [2019] Huyen [2022]. Therefore also making it possible to provide use with better significance estimates for when to stop the experiments.

To give an example of how well this method can work. A Google employee ran an experiment of an ad model, where the performance where coded in for example reasons. He wanted to show that in order to separate an old model which had a click through rate of 1% and the new with 1.05% CTR. In order to provide a confidence interval of 95% using classical A/B testing, would they need 700 000 samples. However with a contextual bandit that uses the Thompson Sampling method, would only around 6 000 samples be needed Rafferty [2020].

[6]LinkedIn - `https://www.linkedin.com/`

So based on this can a method like the contextual bandit reduce the sample size down with 99%, and still deploy a new model with the same confidence. However, using a technique such as bandits require a quick ground truth feedback to make sense. Leading us to the next step.

Model monitoring

Even tho there are a lot of different deployment strategies which was discussed in section (2.1.5), do most of them rely on selecting the best performing model out of a selection. However, this will not be possible to unless some kind of monitoring is added. Or more specifically performance monitoring, as there are a wide range of things that can be monitored.

Such other things would be the system metrics, like CPU, and memory usage. However, this is solved by the DevOps community, and it also provides low value when viewing it from a data scientist point of view.

But as mentioned, the predictions of a model will provide valuable insight. So let us look into what monitoring the input and output of the model can do. This is a fairly easy thing to do, and this information can be used in a wide range of ways. One of the first usages that may come to mind would be to use them for debugging whenever a model produces an odd output. It would therefore be possible to backtrack, and make it easier to digest what and possible why it went wrong.

Furthermore, will this enable new possible like detecting when the data has changed it's natural distribution (Rabanser et al. [2018] Klaise et al. [2020]). Also known as data drift, which will be talked more about in section (2.1.7). But by using either, or both the input and the output can we notify our self that something is wrong. Therefore making it easier to adapt faster to a changing environment.

The next thing we can model is model performance. However, this is hard to do as we need to setup a structure that gets notified when the ground truth value has arrived. At the same time will the problem of latency become clear. As the a short feedback loop could be something from hours to days, while a longer feedback loop could take weeks to months to.

Lastly do also some businesses also compute downstream product metrics. This could be relevant KPIs. And even tho this is the final goal of a good ML system, is this very hard to setup.

Re-evaluate the system

Lastly will the final step be to evaluate the system, and re-iterate on everything. That is often why a lot of people describe the whole process as a infinite loop.

2.1.6 Inference behavior

One essential part to know about the ML product is how it will be used, as this can reduce or increase the complexity of the tech stack significantly.

Batch inference

Usually runs an inference job in the background, using an orchestration tool with a fixed interval. This is often where most companies start, as the tech stack is fairly similar software engineering. However, there might be the introduction of Spark[7] as the processing engine, to handle big data. Such a behavior will often lead to a delay in inference results, potentially making them less valuable. Therefore making it unsuitable for low latency applications.

Real-time inference

Real-time inference solve the latency problem by processing a request as they arrive in a stream such as Kafka[8]. This often leads to latency from milliseconds to a few minutes (Huyen [2022]). Such a system can increase the value of each prediction a lot. However, technical complexity is currently seen as one drawback even though this might be because of a lack of research on streaming architecture. Something that lately has gotten more attention with stream databases, and services like Flink[9].

There is also an interesting concept of online learning where the model is trained in production, instead of being trained on historical data. Therefore, we can make the whole architecture less complex in cases where we have fast feedback loops. One such tool is River[10].

Using a stream processing approach can also reduce operational costs, as we can in some cases leverage lower computing power, as the data is processed in smaller batches. However, this can make debugging a bit harder, as the required tooling is not as common Kleppmann [2017].

Request response

The request response inference pattern works as a middle ground between batch and real-time. Here will the clients request a prediction when they need it, similar to a real-time request. However, this request will be sent through HTTP or gRPC. Furthermore, the request response pattern may not use real time feature, but can leverage features by backfilling features. Similar to the batch inference jobs.

[7] Spark - `https://spark.apache.org`
[8] Kafka - `https://kafka.apache.org`
[9] Flink - `https://flink.apache.org`
[10] River - `https://riverml.xyz/0.14.0/`

Therefore, making our predictions low latency, but potentially using out dated data. Hence, making it very useful for inference where the input data updates with low frequency.

Edge inference

Lastly edge inference. This is often implemented when security is a concern, or when we do not have access to an internet connection. Here is Core ML[11] often used for edge computing within the Apple echo-system.

2.1.7 Data Drift

Nature evolves over time, and so do data. But we can generally categories the drifts in three different ways Klaise et al. [2020].

Covariant Shift

When the distribution of $p(x)$ change, but the distribution of $p(y|x)$ remains unchanged.

This will become clearer with an example. We can use a breast cancer detection model as the relation between $p(y|x)$ is likely to stay fairly constant over time. There are many reasons why such a change could happen. However, let's assume our training data are very skewed. Therefore, we might use a technique as oversampling to make it easier for our model to learn the relationship between $p(y|x)$, or maybe our contain more data about 40 year old women, because of selection bias. Therefore, when we deploy our model will the relationship between $p(y|x)$ still be valid. However, we may use the model on a total different distribution of people. Such as a younger group of woman's.

This may not seem like a problem, as our model still manages to detect the pattern we want. However, we can get misleading performance metrics when evaluating the model offline. Therefore, our model's overall score can decrease in production, while the sub-group performance metrics may stay the same.

Prior Probability Shift

This is when the distribution of $p(y)$ change, but the distribution of $p(x|y)$ remains unchanged.

To present this example, let's continue on the breast cancer example. A prior probability shift, or sometimes known as label shift. Can occur when we have a over-representation, or under-representation of a class in our training set, compared to the real world. Therefore, it is common to have prior probability

[11] Core ML - `https://developer.apple.com/machine-learning/core-ml/`

shift, and covariant shift at the same time. As we often will have a shift in $p(y)$ when we have a shift in $p(x)$. Therefore, the likelihood of being 40 years old will be the same for the training set and real world applications, but the number of cancer detections have changed.

However, it is possible to have a only prior probability shift as well. Let's start by assuming that we are able to train on a data set that reflects the same $p(x)$ distribution in production. After we deployed our model to production was a new drug revealed, that lowered the likelihood of getting breast cancer. Therefore, changing $p(y)$, but keeping the same distribution of $p(x|y)$, as the age distribution for those that got cancer stay the same.

Concept Shift

This is when $p(x)$ stays the same, but $p_{train}(y|x) \neq p_{deploy}(y|x)$. Therefore, it changed the underlying signal, making the previous model useless.

One notable illustration of concept drift can be observed in the domain of breast cancer. This phenomenon occurs when cancer cells undergo evolutionary changes, leading to alterations in their behavior. Consequently, the previously established concept of breast cancer cells becomes outdated and in need of revision.

How to detect drifts

Because data drift can cause huge harm to an ML model had there been a lot of work to try to detect data drift. Again, the method will depend on the problem statement or at with the naive methods.

Some naive methods of detecting data drift in an NLP setting if using something like a BoW Baeza-Yates et al. [1999] could be to notify whenever a new word is not registered in the syllabus. The same thing can be done for categorical values within a tabular setting.

However, this will not work for scalar values, and analyzing one feature at a time can lead to limited detection, as we can not detect multi-variant data drift.

That's why we have seen methods that try to fix this problem in different ways. But training an auto-encoder to generate a latent vector, and then using a multi-variant test looks to be one of the best-performing methods Rabanser et al. [2018].

However, using multiple univariate tests, and then combining the result with a Bonferroni correction do also produces a similar accuracy as the multi-variant-test method. Unfortunately do the same research show that doing no dimension reduction, do perform quite poorly. Thankfully do there exist solutions in order to make drift detection simpler, like Evidently AI[12].

[12]Evidently AI - `https://www.evidentlyai.com`

2.1.8 Data management

Data is the foundational component when creating machine learning systems. That's why data management is essential. If a poor data strategy is at the base can this stop everything.

Schema management

A commonly used method in SQL databases. We which data types to expect in each column, and potentially if it can be null or not. Therefore, we can know with certainty that our data follow a predefine structure, as it is standardized. Such a method can also lead to performance gains, as we can make assumptions about our data. Therefore, compressing or reducing the amount of memory needed.

Columnar semantics

For columnar semantics, will we define a pattern that can be used for our data. Such a strategy can be very useful when we know, or assume that our data should follow a pattern. Examples of such logic would be that an email field contains a @ char, and a domain. Or that an age feature needs to be between 0 and 150. Adding such checks can test our assumptions earlier, and therefore understanding our data in a more correct way.

Multi columnar semantics

Closely related to the previous category. However, here will we check patterns based on multiple columns combined. Leading to the same gains as with columnar semantics, but we cover a wider range of logic. Examples here could be: if first name is defined, will last name also need to exist.

Dataset semantics

Until now have we tested our data on a row by row basis. However, we might want to setup mor advanced tests to check that our data follows a distribution. Here can dataset testing come in handy. We can rather see if our data follows a normal distribution where the mean is within a defined bounds, if there are x

Label management

One of the hardest problems when trying to solve a supervised problem is getting enough labeled data. This is especially the case for image problems, where we need to hire people to hire them manually.

There are a lot of good tools to do this efficiently. Tools like LabelStudio[13], enables users to manually go through samples and label all types of data.

However, this costs a lot of money, but it will also take a lot of time, as we need to wait for the work to be completed. This is why there have been a lot of research in methods in order to get rid of the manual labeling step.

One solution if we have no labels available would be to use *self-supervised learning*. Here will we exploit some property in the data in order to generate unlimited labeled data Chaudhary [2020]. Such a system has show to be extremely powerful, as shown by stable diffusion models. Where the training data is generated by adding noise to an existing image, and finding the function $f(N(x)) = x$ where $N(x)$ is a noise function Suraj Patil [2022]. Such data is shown in the figure (2.4).

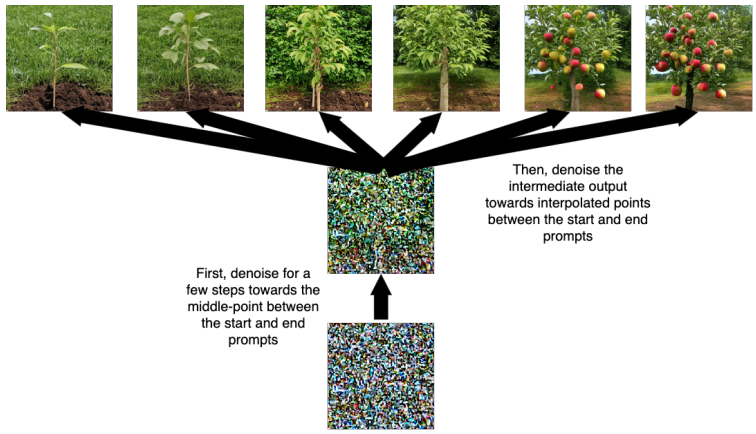

Figure 2.4: Stable diffusion

Another approach that can used when there are a few labeled samples, but a lot of unlabeled samples. Then could *semi-supervised learning* be an interesting use-case, as we can use the existing model to generate new pseudo-labels, and potentially get have a human-in-the-loop for some samples Bach et al. [2018]. Such a method has show to produce equal model evaluation metrics, when using 684K pseudo-labeled samples, or 80K hand-labeled samples. And some tools and services here would be Snorkel[14].

Finally would do *active-learning* also deserve a mention, where the models output tells us which samples to label Settles [2009]. How we would select the

[13]LabelStudio - `https://labelstud.io`
[14]Snorkel - `https://snorkel.ai`

samples can differ a lot, but some methods would be to select samples that are close to the decision boundaries. Another method would be to create an ensemble model, and select the samples where the models disagrees the most.

Therefore do such data management methods have the potential to reduce training costs a lot, as we can use smarter methods in order to increase the value of our training sets.

Point in time data

The examples mentioned in section (2.1.8) was mostly using images and text as input, where there is mostly one source of input.

However, this can often be another case when solving problems in a tabular structure. Here may we need to combine multiple sources into one flat table, and use this as input. But this creates a new problem.

Because it can be hard to know which data is valid at different points in time. One such example could be to know what the average viewing time in the past week have been. However, if we compute the feature from the incorrect point in time, will this effect the model performance. This is why we need to make sure we have point in time valid data, shown in figure (2.5).

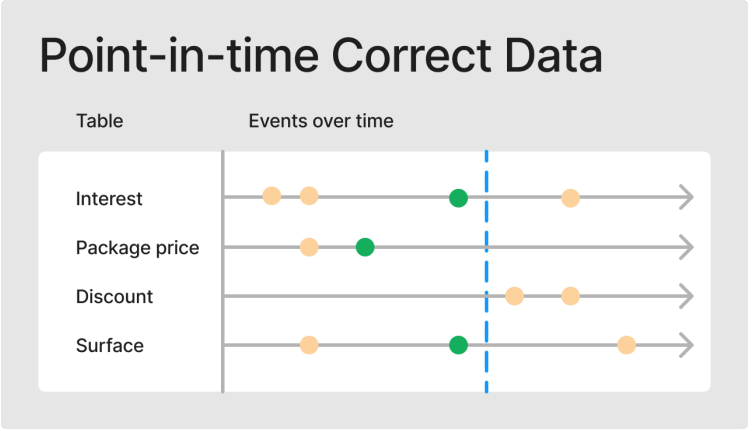

Figure 2.5: Showing how data updates over time, and point in time data needs to select data based on an event timestamp (shown as the blue line). Therefore, we can not use the data after the event timestamp in our model.

This can be an easy mistake to make, and it can be extremely hard to notice, as it can be a silent bug.

Slowly changing dimensions

In order to provide point in time data, is there a need to keep stale data. The strategy used to store the changing data is called a slowly changing dimension and choosing the incorrect one can lead to unusable data. We will mention four different types of slowly changing dimensions, as the concept is the most important thing.

Type 0 is when the data never changes. This is common for data like created at timestamps, as they will not change.

Type 1 and the easiest to implement. This strategy is overwriting old data. Therefore, loosing all previously existing data. This is the easiest to implement in an application and often the most common.

Type 2 adds a new row that contains the updated. We can then determine which row is the valid by filtering on either a flag or the time range it was valid within. This makes it possible to time travel to previous values, but requires some extra filtering overhead when fetching data.

Type 3 tracks only limited history, as the history is stored in a new column.

Log and wait

Setting up slowly changing dimensions can require changes to the data infrastructure. Therefore, it can be costly to setup. Hence, log and wait can simulate a similar approach to slowly changing dimensions in ML systems, but with some reduced flexibility. By logging all the features used for inference, together with the timestamp, we can log all our features to a data storage at different point in times. Therefore, we get the same effect as SDC, but with less infrastructure. However, using log and wait means we can not change the input features, but only leverage the features available at inference time.

Training Serving Skew

After a model has been deployed in production is there still a chance of silent failures. One such failure that can easily go undetected is a training serving skew. This is when a model works very well in the training phase, but horrible in the serving part. This can often happen because the input data to the online model is different in nature.

Maybe because of pre-processing logic being out of sync. One such example would be using the incorrect mean or std in a standard scalar transformation.

Feature Stores

To solve the pain points described in section (2.1.8), ML engineers often use a component called a feature store. Which improves data quality. Here will they

often provide a point in time correctness. Consistent features between offline, and online processing, while also choosing the best storage based on the situation. Some solutions also provide a data catalog, and feature versioning, making it the go-to place for feature shopping. One feature store structure is shown in Figure (2.6), where the feature definitions is stored in a shared storage, and then adapts to the use-case. Some solutions are Tecton[15], Feast[16], AWS SageMaker[17].

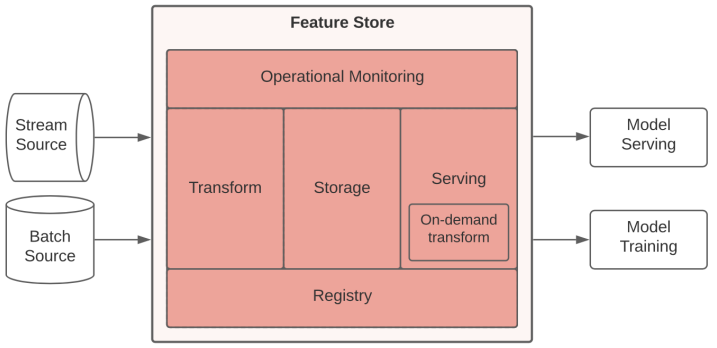

Figure 2.6: Feature Store - `https://www.tecton.ai/blog/what-is-a-feature-store/`

Evaluation Store

The feature store focus on improving the quality of our model input, or the X. However, when we have deployed a model will it be interesting to have a method of measuring the performance of such a model. This is what an evaluation store will do. It records the predicted values, the ground truth, and in some cases the input values as well.

Therefore, making it possible to monitor model performance, sub-group model performance, and in some cases debug the predictions based on the input of the models.

Data Lineage

Understanding how data flows through a system can be hard as the system grows. Therefore, the concept of data lineage have been extremely helpful in big data

[15] Tecton - `https://www.tecton.ai`
[16] Feast - `https://feast.dev`
[17] AWS SageMaker - `https://docs.aws.amazon.com/sagemaker/latest/dg/whatis.html`

applications. Making it possible to see each step in the data pipeline, and how they integrate into each other, as shown in figure (2.7). However, not all tooling are able to capture where the sources stem from and how they transform. Therefore, making it harder to find and debug errors.

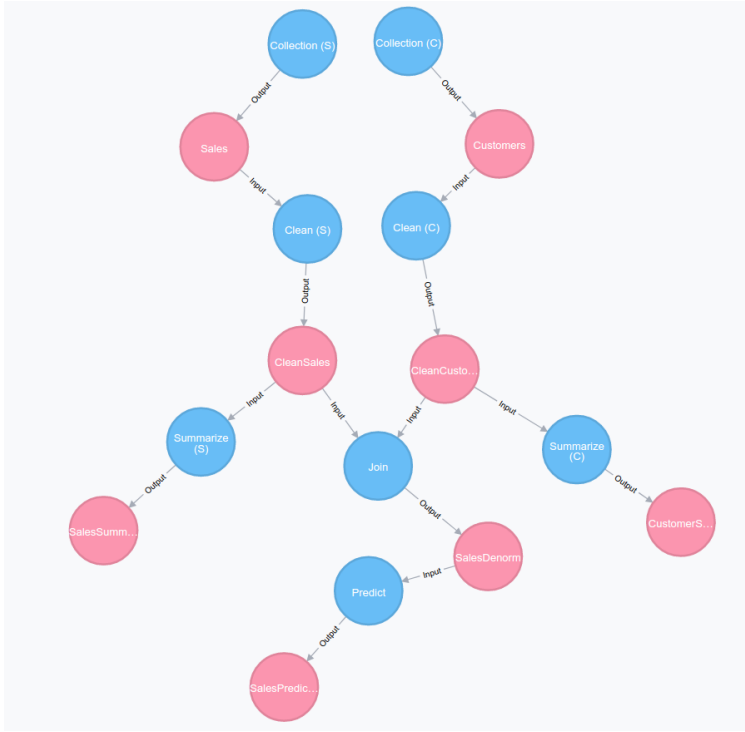

Figure 2.7: An example of data lineage

2.1.9 Automation

One of the core principles of DevOps is to make the developer care more about the operation as well. In practice is this often done through automating processes. Automation can therefore help with catching problems earlier in a production environment.

However, extra automation are helpful to machine learning products, and some of them will be introduced in this section.

Continuous integration

Continuous integration (CI) is a common practice for software engineering, as it quality checks our work. Therefore, implementing CI into machine learning can vastly improve our products. Setting up a pipeline that quality checks the work of the data scientist, ML Engineer, or whoever is updating the system is always a good idea.

For instance, unit tests might quality-check pre-processing pipelines and transformations. For instance, will NaN values or dividing by 0 be handled correctly?

Furthermore, data scientists often focus on research and the logic behind the model. Which can lead to less focus on code quality. Therefore, setting up linting can help make the project more maintainable for future changes. Such a pipeline could be combining flake8[18] together with Github Actions[19] for distributed CI, or pre-commit[20] for a CI pipeline that runs on commit changes.

Continuous deployment

Reducing the work needed to deploy new software is another known practice from DevOps, known as continuous deployment (CD). Such tools makes it possible to focus on smaller releases, which again makes it easier to find bugs, and focus on increasing the value of the product.

Within a machine learning context. For a data scientist will CD reduce the focus on learning technologies like Docker[21], and kubernetes[22]. Which can be a steep learning curve, and unwanted focus area for the data scientist role.

Continuous Training

In order to tackle the challenge of data drift have a concept of continuous training emerged. This is an extension of the continuous integration (CI) and continuous deployment, within the DevOps movement. A process that trains a new model will automatically start when data drift is detected, and therefore improve on the old model. Here could something like AirFlow[23] be used to trigger a new training pipeline.

[18] flake8 - `https://flake8.pycqa.org/en/latest/`
[19] Github Actions - `https://docs.github.com/en/actions`
[20] pre-commit - `https://pre-commit.com`
[21] Docker - `https://www.docker.com`
[22] kubernetes - `https://kubernetes.io`
[23] AirFlow - `https://airflow.apache.org`

2.1.10 Versioning

Code versioning

Versioning code is a known practice in software engineering. This practice helps improving reproducibility, and fault-tolerance. Therefore, we will make it easier to diagnose bugs, but also faster to revert to old versions. Often done through Git[24]. However, when deploying ML will even more be needed to version.

Data versioning

As described in section (2.1.1), the logic of a machine learning models are defined by their training data set. This makes the data set valuable to debug, whenever a problem occur. Like explaining why we have a sub-group bias.

Therefore, a data snapshot needs to be stored somewhere, which means another artefact to version. So a naive solution would be to store it in git. However, Git is not designed to version large file, and can at most upload 2GB per commit, if Git LFS[25] is used.

That is why versioning tools like DVC[26] exists, which tries handles large files by creating a pointer to a storage platform.

However, a system using a slowly changing dimension can fulfill a lot of the needs of a git like data versioning system[27].

Model versioning

In section (2.1.5) we mentioned different types of deployment strategies. Such strategies will not be possible without supporting model versioning. This is often implemented by defining a storage where the can easily be fetched. Therefore, a simple solution of this would be to store the model in AWS S3, or adding something more advanced like MLFlow[28].

2.1.11 Declarative design

In the previous years have there been an increase in declarative frameworks. This is mostly clear within the front-end community, where tools like React[29], Flutter[30], and SwiftUI[31] have become the go-to tool for UIs.

[24] Git - `https://git-scm.com`
[25] Git LFS - `https://git-lfs.github.com`
[26] DVC - `https://dvc.org`
[27] `https://locallyoptimistic.com/post/git-for-data-not-a-silver-bullet/`
[28] MLFlow - `https://mlflow.org`
[29] React - `https://reactjs.org`
[30] Flutter - `https://flutter.dev`
[31] SwiftUI - `https://developer.apple.com/xcode/swiftui/`

Furthermore, SQL[32] the most well known tools for managing data is declarative. Enabling the user to define how to managing data, without defining how it is done. Therefore, making it use-full in multiple scenarios, as it fits both transaction, and analytical use-cases.

Declarative APIs have therefore proven to provide a good developer experience, while still being flexible.

This is why we have started to some products that try to make the whole deep learning process declarative with tools like Ludwig[33]. However, Ludwig is based on YAML[34] file which do not provide type safety, as it is a schema on read format Kleppmann [2017]. Therefore, not fulfilling all DevOps practices.

2.1.12 Model cards

One of our most valuable questions in the interview process where. *According to Sculley et al. [2015], is there a lack of good ML abstractions, so what do you find to be the best for ML?*. This question lead to a mention of model cards.

The goal of model cards are to provide transparency, and trustworthiness Mitchell et al. [2019]. Therefore, a model card describe how a model has been created, which data is for training and evaluation, when it was created, and the reasoning behind the architecture. Furthermore, performance metrics, usage-guides and ethical considerations will also be described, as seen in figure (2.8).

Tools as Model Card Toolkit[35] defines a clear process for setting up model cards.

2.1.13 Vector database

The rise of LLMs and embeddings have created a need for new types of databases, as classical databases have not been built for similarity search. Therefore, there have been a rise of vector databases where it is possible to do semantic similarity search on unstructured information.

With the recent advances in LLMs, have vector databases been a valuable tool to simulate long term memory. Because our vector databases can find related information based on the users query. Therefore, help the LLM as we can perform in-context learning.

Some examples here are Pinecone[36], but also key value stores like Redis[37],

[32] SQL - `https://www.iso.org/standard/63555.html`

[33] Ludwig - `https://ludwig.ai/latest/`

[34] YAML - `https://yaml.org`

[35] Model Card Toolkit - `https://ai.googleblog.com/2020/07/introducing-model-card-toolkit-for.html`

[36] Pinecone - `https://www.pinecone.io`

[37] Redis Vector Search - `https://redis.io/docs/stack/search/reference/vectors/`

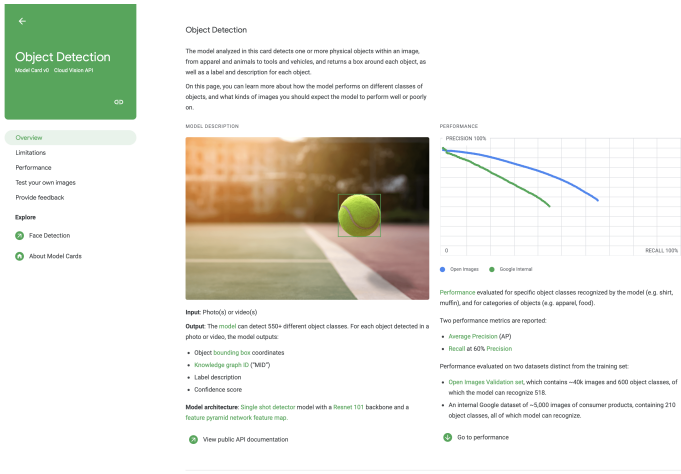

Figure 2.8: Model card from Google

and Cassandra[38] have added such features.

2.1.14 Data centric AI

In the recent years have a new approach to ML been growing in popularity. Namely data centric AI. They argue that ML models have started to stabalise. Therefore, imporoving data quality can have a greater impact on model performance.

Therefore, one data centric method would be to find and remove falsely annotated data points in a data set. Resulting in a clearer signal, and leading to improved model performance. Northcutt et al. [2022] found that using a technique called confidence learning outperformed all recent model centric techniques on the CIFAR data set, when simulating noisy y labels, and missing y labels.

Self-supervised learning

A common way to collect annotated data, have been to show some input, and manually collect the ground truth value. However, *self-supervised learning* tries to collect data the other way around. By collecting the intended output, and manipulate the output so we can produce an input.

[38] Cassandra - `https://cassandra.apache.org/_/index.html`

Noise	20%				40%				70%			
Sparsity	0	0.2	0.4	0.6	0	0.2	0.4	0.6	0	0.2	0.4	0.6
CL: $\boldsymbol{C}_{\text{confusion}}$	89.6	89.4	90.2	89.9	83.9	83.9	83.2	84.2	31.5	39.3	33.7	30.6
CL: PBC	90.5	90.1	90.6	90.7	84.8	85.5	85.3	86.2	33.7	40.7	35.1	31.4
CL: $\boldsymbol{C}_{\tilde{y},y^*}$	**91.1**	**90.9**	**91.1**	**91.3**	86.7	86.7	86.6	86.9	32.4	**41.8**	34.4	34.5
CL: C+NR	90.8	90.7	91.0	91.1	**87.1**	**86.9**	**86.7**	**87.2**	**41.1**	41.7	39.0	32.9
CL: PBNR	90.7	90.5	90.9	90.9	**87.1**	86.8	86.6	**87.2**	41.0	**41.8**	**39.1**	**36.4**
INCV (Chen et al., 2019)	87.8	88.6	89.6	89.2	84.4	76.6	85.4	73.6	28.3	25.3	34.8	29.7
Mixup (Zhang et al., 2018)	85.6	86.8	87.0	84.3	76.1	75.4	68.6	59.8	32.2	31.3	32.3	26.9
SCE-loss (Wang et al., 2019)	87.2	87.5	88.8	84.4	76.3	74.1	64.9	58.3	33.0	28.7	30.9	24.0
MentorNet (Jiang et al., 2018)	84.9	85.1	83.2	83.4	64.4	64.2	62.4	61.5	30.0	31.6	29.3	27.9
Co-Teaching (Han et al., 2018)	81.2	81.3	81.4	80.6	62.9	61.6	60.9	58.1	30.5	30.2	27.7	26.0
S-Model (Goldberger et al., 2017)	80.0	80.0	79.7	79.1	58.6	61.2	59.1	57.5	28.4	28.5	27.9	27.3
Reed (Reed et al., 2015)	78.1	78.9	80.8	79.3	60.5	60.4	61.2	58.6	29.0	29.4	29.1	26.8
Baseline	78.4	79.2	79.0	78.2	60.2	60.8	59.6	57.3	27.0	29.7	28.2	26.8

Figure 2.9: Accuracy using conficence learning for different levels of label noise, and sparsity

Such a process have been way more effective, as we can automate the data collection process, and get higher quality data.

Semi-supervised learning

Sometimes known as weakly supervised learning tries to somewhat solve the same issue of missing label data, but in another way. Rather then generating the input will semi-supervised estimate the missing y value using algorithmes. Such methods could be based on the confidence of previously trained model, as we can assume that the model will mostly be correct on confident answers. Another approach is to setup an ensable model and set the label to the majority vote. Lastly, have a popular approach been to setup heuristics labeling functions.

The semi-supervised have been super successful, and have lead to the creation of dedicated companies such as Snorkel.ai[39].

Active learning

Selects the datasets that provide the most amount of value to learn from. E.g. Select points close to the decision boundary, in order to learn nuances faster.

Can use active learning to do this? Active learning will at least provide the samples that we will have the most value from when labeling them. However, can this be used to select better data as well?

Active learning have different strategies to select data. - Uncertainty - Variance reduction - Loss reduction

[39] Snorkel.ai - `https://snorkel.ai`

Loss and variance reduction are computational heavy, as we need to train a new model for each datapoint label combination. However, this might still be valuable for use-cases where the model provide high value (Ref, medical care).

We also have different uncertainty strategies.

Select top n based on uncertainty (Batch)

Sort a dataset based on the uncertainty metric, and pick the top n records. Such an approach is useful for batch processes, when we have access to the finite amount of records. These records will in theory provide more information around the decision boundary.

Select values under a threshold (Streaming)

Select all values that are under a certain threshold, and add them to a dataset. This will in theory select the datapoint the are unsure about.

Such a strategy are useful for streaming systems, as a records can be evaluated individually

Select records with lowest difference between first and second label

Compute the uncertainty, and find the labels with the least difference between the most, and second most confident prediction. Labeling such datapoints will improve the decisions boundary around closely related labels. Again we can use a streaming or batch approach.

Down sides of using uncertainty

Deep learning models tend to be fairly confident in prediction that is outside of the training set. Therefore, uncertainty can be an unreliable metric to use. Such a metric will also only be valid for tasks where an uncertainty metric can be provided. Which is not always the case. Ref. Regression tasks (Maybe a confidence interval can be provided though).

Loss and variance reduction

Select the datapoints that reduce the most amount of variance in the model. Since this will be done on data where we do not know the ground truth, is there a need to guess all labels, and train a new model.

Curriculum learning

Another interesting approach inspired by how human behave is curriculum learning. Bengio et al. [2009] shows how order of samples can effect the time to convergence, and performance of a model. Therefore, enabling the model to learn high level concepts first, and then increasing the difficulty can have a positive effect. However, one challenge with curriculum learning is how to decide how difficult a sample is.

2.1.15 LLMs

The rise of LLMs and ChatGPT have changed how we can create applications. Models like ChatGPT needs way less examples to train a well function model. Furthermore, by testing on a few hard examples can it make the model robust.

We can setup LLMs where sub tasks will be a classification task. Therefore, we can use the proposed solution to evaluate the sub tasks.

Measure "We like it", "it was probably liked", "it was incorrect", "Was correct (in cases where there are complex results)", "setup a set of evaluation criterias (aka. use the LLM as a supervised model, either in the form of one hot encoding, ordinal encoding etc.)".

Figure 2.10: Supervised vs Prompt based ML life cycle

2.1.16 Insights from Interviews

At the start of this master thesis, interviews were conducted to better understand the complexities and hurdles associated with deploying Machine Learning models. These interviews were performed with Senior ML Engineers and Architectural Leads within the ML domain. The template used for these interviews can be found in the appendix.

These discussions yielded several intriguing insights.

Unique Challenges in ML

All the interviewees identified data as one of their primary challenges. Moreover, a group among them quantified that approximately 60% of their issues originated from data-related errors. These challenges encompassed various aspects, such as the fusion of data from disparate sources, loading data annotations, ensuring point-in-time validity of our data, and maintaining data consistency across different environments and machines.

Model Validation

The process of validating that a newly trained model performs as anticipated was another commonly cited pain point. All respondents expressed that the traditional train, test, validation splits were often inadequate, leading to mistrust in locally computed validation metrics.

Therefore, the need for real-time validation even for batch models was emphasized, as this would provide a more accurate comparison between local metrics and real-time ones. Furthermore, the respondents indicated that they frequently utilized a custom-made dataset for model validation. This dataset was separate from the classic train, test, validation split and was more akin to a regression test set or an integration test set.

Abstraction

Upon querying about known effective abstractions used for ML, it was evident that most of these were at a low level and necessitated substantial infrastructure knowledge. Kubernetes emerged as a popularly employed technology, but it is known for its complexity and steep learning curve. ML technologies have been developed to harness Kubernetes, such as KServe. KServe was commended for facilitating the establishment of a model graph, enabling the linking of different models. However, it was noted that there exists a significant interdependence between software solutions and ML models, which could potentially lead to a lock-in situation.

Organization

Furthermore, there was a common theme that the business organization had a huge effect on the success of ML products. It could often be difficult to green light the work needed to reduce technical debt or enhance ML Operations practices. This made it more challenging to monitor the success of a model. Furthermore, such an effect could be exacerbated by the lack of constraints on an ML project. Metrics for success were sometimes loosely defined, rendering the success criteria subjective.

2.2 Hidden technical Debt in ML Systems

Technical debt is a metaphor used to describe long term costs by prioritising to move quickly in software development Sculley et al. [2015]. Taking on technical debt can be very useful to show the potential value gain of a solution. However, we need to pay down that debt eventually. Either in the form of refactoring, writing more tests, deleting dead code or change the architecture.

Therefore, the goal of paying down technical debt is not to add new features, but rather to make the existing software more flexible, error resilient, and easier to understand. Furthermore, technical debt can be hard to identify, which makes it easy to grow over time.

ML systems suffer from the same types of technical debt as software development. Furthermore, they also have an increased complexity because of the added needs as described in section (2.1.4). However, ML systems tend to be developed at a system scale, and not an isolated code level basis. Therefore, the paper argues we do not have the correct tooling or abstractions to describe, and pay down such system wide technical debt.

The paper also categorises different technical debts. We will cover 13 of the most relevant categories for this project.

2.2.1 Entanglement

We often design systems that manage a sub-set of problems. One such system could be an ensemble model, where each model have errors in their own domain, but they perform well when combined with others.

However, such a system can quickly lead to what the paper refers to as CACE, or Change Anything Changes Everything. Therefore, if adding a new feature X_n helps to improve one underlying model, can it lead to a decrease in the overarching model.

2.2.2 Correction Cascades

A very similar problem can occur in the correction cascades problem. However, rather then introducting an new feature X_n, can it be tempting to improve the performance of one model by using the output of another model. Therefore, if we have a model m_A trying to solve the problem A. However, we might have a related problem A'. Therefore, it could be tempting to make the model $m_{A'}$ use the output of model m_A. Furthermore, we might get a new problem A'', and now we have a set of dependent models.

As a result, we quickly create a system where an improvement in our first model can reduce the performance in the second step and so on. Furthermore, it can make it harder to train a new model, as we have a pipeline of models, rather than one model to train.

2.2.3 Unstable Data Dependencies

In the same way that our code have code dependencies, will our data have data dependencies. However, the paper argues that finding such data dependencies is way harder then code depencencies. The reason being that our data depencencies are not as clearly defined, and our tooling struggles to do static analytices on the data depencency graphs.

Therefore, one such debt is unstable data dependencies. Leveraging unstable data signals can quickly happen, when it is unclear what the data is intended to be used for, and when it is maintaind by another team. Therefore, changes can be implemented that breaks our assumptions, making them unusable for our ML models. One such example could be the usage of an ordinal encoded value. Another team may find such a value useful and add it to a model. However, after some time may the team maintaining the ordinal value change it to support a wider range of values. Therefore, the underlying semantic meaning of the feature will change, and potentially degrade the ML model that used the value.

2.2.4 Underutilized Data Dependencies

A number of features used in a model can quickly grow over time. Therefore, we can end up in a situation where only a small fraction of the feature provide any useful information. Very similar to how code dependencies can end up being unused over time.

There are multiple reasons for why data dependencies can be unused over time.

Legacy features Over time will data evolve. Therefore, data may change how it is stored making older features redundant. However, it may be unclear which features are out of data, making it hard to detect legacy features.

Bundled Features We sometimes test a bundle of features at the same time to see if they have an effect. However, because of time constraint could we end up adding all of then, even though it is only a few that have an effect. Therefore, setting up unneeded data dependencies.

ϵ-Features A Machine Learning researcher have a goal to create the most performant model. However, sometimes it can be hard to evaluate if the improved performance is big enough to make it worth the added complexity gain.

Correlated Features We often have multiple features that are correlated with each other. Therefore, it can often be good enough to only add one of them. However, it can be unclear which one to use, leading us to potentially using the non-causally one. Therefore, leading to a more brittle system.

2.2.5 Static Analysis of Data Dependencies

Having analyser tools like mypy[40], and compilers for type safe languages can help find errors, and optimise code. However, similar tooling for data is not as common, but such a tool can find similar errors, find unclear data consumers, and notify about unsafe data migrations. Therefore, a tool that analyse our data dependencies could provide emense value at a system wide level.

2.2.6 Glue Code

The ML model is the underlying engine of the whole system. However, it is surprising how little code is related to the actual model. As shown in figure (2.11). Therefore, it is common to implement code to fill the layers between each component. Sometimes known as plumbing. However, such plumbing can be an anti-pattern, making it harder to maintain ML system as they grow in size and complexity.

Glue code is one such anti-pattern, where we write code to translate data from and to different generic data formats. Such code often arise when using open-source ML software developed in isolation. Therefore, making it harder to integrate in a bigger system, as translation code is needed. Therefore, the paper argues that some ML systems can end up being 5% ML code, and 95% glue code.

2.2.7 Pipeline Jungles

As our ML pipelines grow in complexity will the different paths in our pipeline also become more complicated. Therefore, it can become an intricate web of

[40] Mypy - `https://github.com/python/mypy`

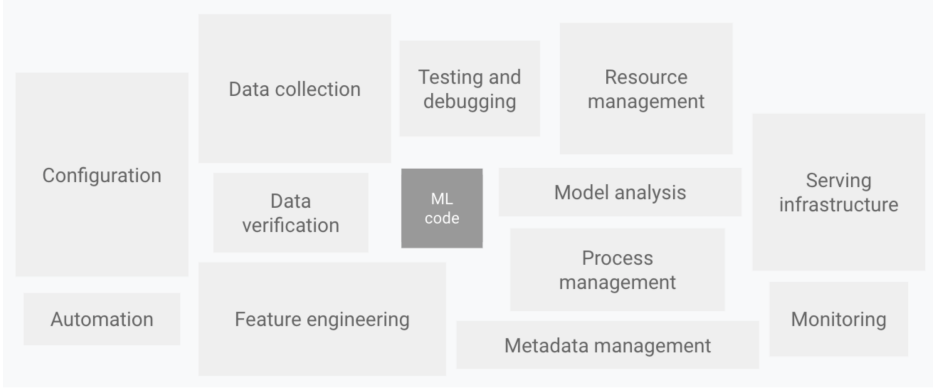

Figure 2.11: Different components in an ML system

fetching, joins, and sampling. Leading to a codebase that is hard to manage, and costly to test.

2.2.8 Dead Experimental Codepaths

A consequence of Glue Code and Pipeline Jungles can lead to the need of refactoring. However, because of the complexity can we end up with dead codepaths. Therefore, we risk maintaining an experimental code path that could have been deleted, or provide low value.

2.2.9 Plain-Old-Data Type Smell

ML tend to operate on plain old data types, such as integers, and floats. However, our data can be represented in a richer manor. Therefore, by representing our data pipelines using plain old data types can lead to technical debt. Instead we should represent our data using higher level concepts as coordinates, emails, urls etc. Therefore, making it possible to extract more information out of what otherwise would be a raw float or string.

2.2.10 Multiple-Language Smell

An ML system often spans a wide range of components. Therefore, we may tend to use multiple languages in different components. However, such a design leads to an increased load on the developers as they need to know a wider range of languages and need more experience. Furthermore, multiple languages can lead

to friction in hiring new personnel, because of the extra requirements. Therefore, reducing the number of languages makes it easier to maintain in the long run.

2.2.11 Prediction Bias

When we have a system that work as intended should it usually predict a similar distribution to the ground truth distribution. However, this is not always true if we train our model incorrectly, or when the model is outdated, potentially because of concept drift as mentioned in section (2.1.7). Therefore, frequently checking the predicted label distribution against the expected distribution can be a good practice to make sure our models represent the real world.

2.2.12 Data Testing Debt

Software 2.0 shown in figure (2.2) use data to define the behavior in ML systems, rather then code. Therefore, makes it sense to test our data, in a similar way that we test our code. Therefore, making sure our data follows our assumptions, and the distributions we expect will be needed. Furthermore, the practices described in section (2.1.8) are examples of potential methods.

2.2.13 Reproducibility Debt

Debugging our code path, and our predictions will eventually be needed. Therefore, enabling us to test previous runs, or reproduce our system state is essential. However, we often have randomness in our pipelines. Either in our hyperparameter search, training, test split, or data selecting. Therefore, it is important to make sure we make our whole pipeline as deterministic as possible, resulting in a reproducible program.

2.3 Motivation

This section will cover the motivation for starting on the described project.

2.3.1 New point of view

As shown in chapter (2), a wide range of knowledge is needed to deploy ML products. However, a new point of view about ML systems was presented when combining the theory of data centric AI section (2.1.14), and the existence of the debt described in section (2.2).

Using the knowledge learned from data centric AI could we move away from thinking how to solve the different components in figure (2.11). Rather we group

our problems related to an ML system into three high level buckets based on viewing ML systems as the formula $f(X) \Rightarrow y$.

2.3.2 The model $f(X)$

The first set of problems relate to the model it's self.

Model Creation

This problem is all about creating a model that learns some pattern. Therefore, this can be seen as the classical approach of finding the best algorithm. Therefore, this is where setting up neural nets, decision trees, and all other algorithms fits.

Model Exposing

Furthermore, a model will not provide any value if it can not be used by the end user. Therefore, exposing it either through an inference server, storing it on the edge, or trigger it through a streaming process. As a result, this is where technologies like KServe, OpenAI's API, and to a degree Huggingface comes into play.

Model Registry

Unfortunately things will not always go as planed, so being able to track how our $f(X)$ evolved is needed. Therefore, technologies like MLFlow, can be used to see each $f_t(X)$.

Model Summary

Furthermore, knowing how a model is intended to be used, it's shortcomings, and it's pros is important. Therefore, a model card can be used to summarize each $f_t(X)$, making it clearer how it should be used.

Experiment tracking

However, a summary is not always good enough for everyone. Therefore, being able to view how $f_t(X)$ came to be is important. Providing a log of metrics as our model is trained, in addition, can all kinds of evaluation graphs be logged - as confusion matrices, and log which features contribute the most to our output.

2.3.3 The input X

However, we are not able to create our model without any data. This subsection will discuss the problems related to input data. There are some overlaps with the the input and output for some of the challenges. However, the choice to split input and output is two fold.

First, even though supervised methods is what this paper have the most focus on, is it important to keep in mind that unsupervised methods also exist. Therefore, will we not have a ground truth to train for, but we will still need to maintain our output as a separate artefact.

Secondly, the way we handle our output of a model differs from how we handle our output. Therefore, will they have slightly different needs.

With this said, some of the following problems will handle both X and y for supervised problems.

Creating Data Sets

First of all, we need data to train our model. Therefore, the first problem will often be to find, and load data from some one or more sources. This will be the case if using standard supervised methods, but also for self supervised and semi supervised learning methods.

Data Cleaning

Furthermore, our data can be unclean and messy. Therefore, a common step is to clean our data. Either removing outliers, confirm that data exist within reasonable assumptions, and so on. Some checks here could be checking upper, and lower bounds, the number of missing values and so on.

Furthermore, semi-supervised learning can be leveraged here when creating a training set, to increase the number of labeled data points is missing.

Data Versioning

Replicating our data is important, as it is needed to make sure our models are replicatable.

Here can a wide range of methods be used, but the one important thing is that the data schema remains the same. Therefore, if we use log and wait, slowly changing dimensions, or a tool like DVC is irrelevant. But the important thing is that they follow the same schema, with the same processed features, and data validation.

Online Data Serving

When a model have been deployed, will they need to have some input features to infer on. What the features are will again not be that important in this case. However, we need to make sure that the features are in the same schema, and that they keep the same position in our input vector.

Therefore, a feature store can again help with ensuring such a requirement. However, choosing the incorrect data format can lead to high latency, making the model useless. Therefore, evaluating different data storage method are important to ensure an usable product.

The Training Serving Skew

We can get new problems when our ML system use different sources for our historical data, and the inference data. Something that is common with the inference server design. Therefore, having an shareable, unified way of loading and transforming data can help with the training serving skew.

Monitor Features

Lastly, previous problems have not considered if the model is used on data that's within the training distribution. Therefore, by setting up detection method on the input data, can be sure if use the model in the correct manor, or if we would need to retrain the model.

2.3.4 The output y

Finally, the most important artefact that ML offers. The output y.

Evaluation

As mentioned in section (2.1.5) will the relevant evaluation metrics change based on the ML use-case. However, the end goal will still be to evaluate how good our $\hat{y}$ will be. Either by comparing to a ground truth value y, or for unsupervised could we setting up distance metrics to evaluate if the $\hat{y}$ follows the intended results.

Furthermore, more advanced ML systems might compare our $\hat{y}$ against special data sets to check behavioral tests Ribeiro et al. [2020].

Monitoring Predictions

Similar to how the input was monitored could the same thing be done for the output. However, monitoring the output will potentially catch a different type

of data drift.

Explanations

Being able to explain why the model predicted it's results is important to build trust. Such a property have show to be especially important for critical applications.

2.3.5 Implications

Changing how we view MLOps have interesting implications.

With the use of our classical view of MLOps in figure (2.11) will we increase our surface area of complexity. More precise, we think about functionality. Therefore, leading us too potentially setup and manage one new service per component. However, we need to validate the same data schema, and data semantic across services. Therefore, we make our data logic a bi-product out of our functional implementation, as shown in figure (2.12). In other words, we will have a complexity of $O(n)$ per functionality we want for our model.

However, we can reduce the complexity down to $O(1)$, by thinking about ML models using the $f(X) \Rightarrow y$ formula. Furthermore, by describing how our data behaves without the implementation details will it unlock a wide range of possibilities. Hence, handling our ML components (X), $(f(X))$, and (y) as artefacts can we automatically setup functionality while keeping the semantics consistent across separated services. Therefore, leading to a system where the data logic defines the implementation, as shown in figure (2.13).

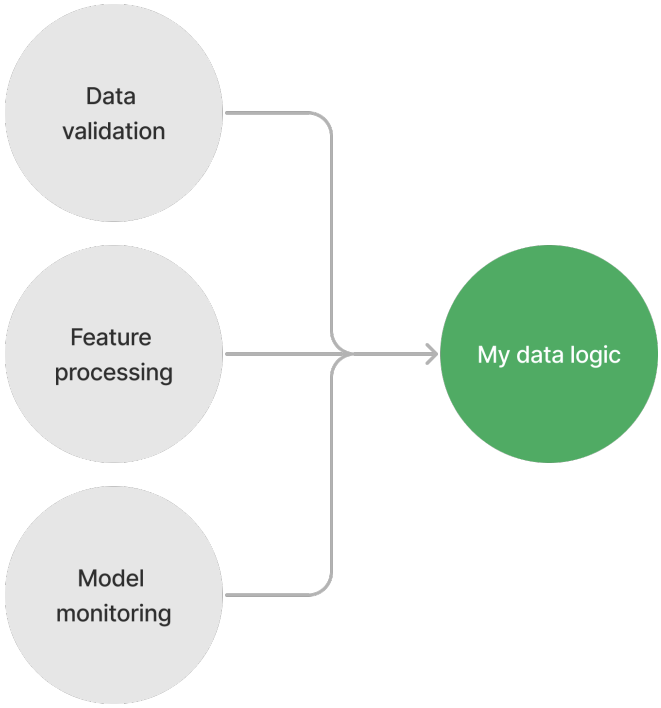

Figure 2.12: How code derive data logic implicitly at a system scale

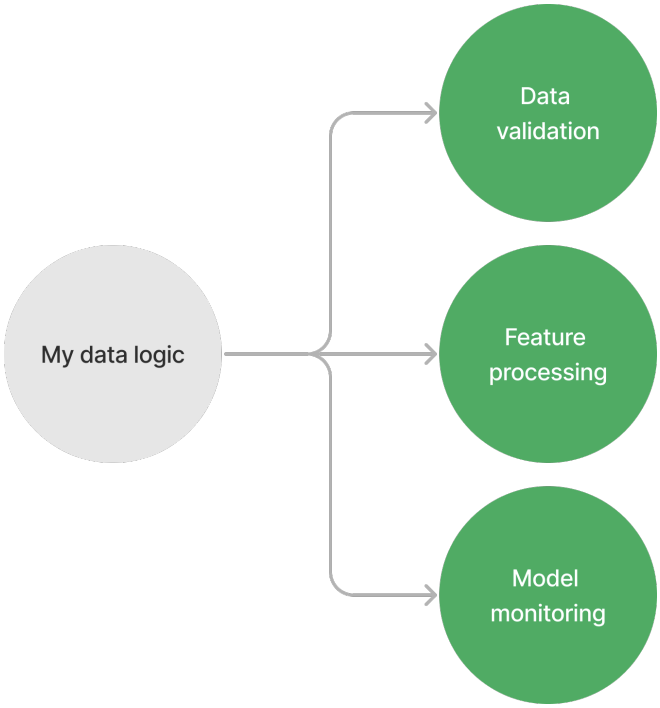

Figure 2.13: How data logic can generate code to unify logic at a system scale

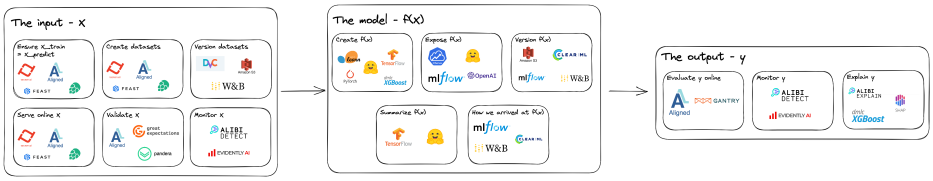

Figure 2.14: Technologies grouped based on what they solve in the equation $f(X) \Rightarrow y$ Mollestad [2023]

Chapter 3

Proposed System

This chapter provides a description of the proposed system, its core principles, and functionality.

3.1 Core Design Choice

As discussed in section (2.1.4), setting up a machine learning (ML) system involves numerous components. However, when considering that ML revolves around the relationship $f(X) \Rightarrow y$, it becomes unclear why such a diverse range of components is required.

Furthermore, categorizing components based on whether they address issues with X, $f(X)$, or y helps reveal that many dependencies stem from the model interface rather than the underlying model itself. Consequently, expressing ML systems in terms of data logic rather than code may prove more effective. The rationale is that most code can be derived from data logic (see figure 2.13), whereas maintaining a code base that adheres to the same data logic is considerably more challenging (see Figure 2.12).

At its core, the proposed solution endeavors to introduce a new approach to designing and describing machine learning applications. Instead of writing code that defines the implementation of the ML system, Aligned aims to reverse this approach. By describing the system's data logic at a high level, it becomes possible to generate the implementation based on this logic.

Furthermore, Aligned makes it possible to keep functionality and documentation together. Therefore, reducing the risk of out dated code and data catalog documentation.

Consequently, the solution consists of two main components: the metadata compiler and the implementation compiler, as shown in figure (3.1).

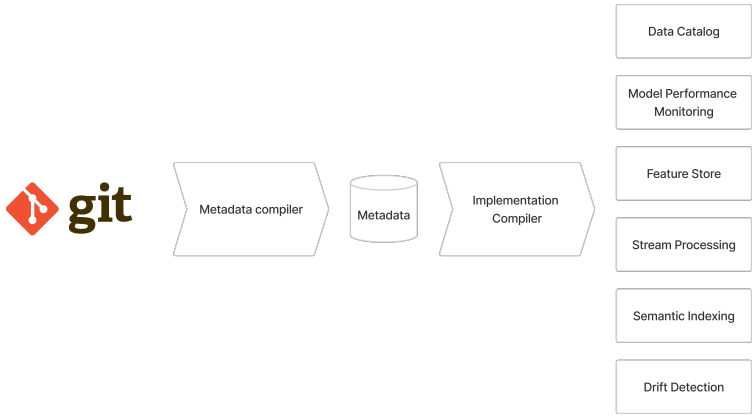

Figure 3.1: Aligned's components presenting how code gets compiled down to metadata, and then used to generate functionality

The described solution may bear some resemblance to a feature store. However, a feature store only model the input X. Therefore, the added possibility to describe model predictions ($\hat{y}$) and ground truths (y) makes the proposed solution more generic, and enable interesting new functionalities.

Moreover, by combining output modeling and data lineage, it becomes possible to track the flow of data throughout the entire system, even when multiple models are interdependent. In contrast, a feature store solely models the input (X).

3.2 Metadata Compiler

Representing the entire application based on data logic requires collecting a substantial amount of data. However, as the reader will discover, the volume of data necessary to collect makes manual writing inconvenient. Hence, a crucial aspect of the system is providing a set of tools that can elegantly and comprehensively describe the system's metadata.

3.2.1 Data Source

A data source describes any form of data, ranging from a database table, key-value store, file, or structured folder. Consequently, the system can support existing infrastructures such as application databases, data warehouses, as well as more complex infrastructures like data lakes.

Furthermore, data sources will vary in the amount of needed metadata. Therefore, each data source is represented with a unique key enabling us to understand which metadata to decode. For instance, a CSV data source can contain a separator symbol and a compression format. However, a PostgreSQL source needs a table name, and potentially a schema name. Hence, enabling the user to be flexible, yet detailed.

3.2.2 Feature View

To extract value from data sources, it is necessary to describe their format. Hence, the concept of a feature view emerges, which can be considered as a collection of features.

Consequently, a feature view encompasses the information outlined in the forthcoming sections.

Entity

Each feature is associated with a particular identifier. For instance, features related to a person, such as age, height, and weight, can be associated with either a name or personal number. Thus, the entity describes which data is used to identify the remaining features.

Therefore making it possible to ask for the age given user named x, or for the increase in weight for the last 30 days for user named x.

Event Timestamp

Features are often subject to updates over time. To facilitate retrospective analysis and select features available at the intended prediction time, it is crucial to identify the timestamp to filter on. This is known as the event timestamp.

Therefore, making it possible to filter out features that was non-existing at inference time, or compute aggregations in a time window.

Feature

Accessing features is among the most critical aspects of the system. Therefore, Aligned simplifies the definition of a data schema, incorporating validation and documentation.

Additionally, while raw data types are typically described, higher-level data types like emails, coordinates, and URLs can also be specified. This facilitates the inclusion of validation and the standardization of common transformations.

Derived Feature

Even though a feature contains the raw inputs, will a derived feature describe a feature that depends on any other features. One such example could be the host name in an URL. Because of this will a data lineage be needed. This means that the compiler needs to collect which features it depends on, which transformations to perform, and what the output data type will be.

Derived features describe those that depend on other features. For instance, the host name in a URL represents a derived feature. Consequently, the compiler needs to collect data lineage information, including the features it depends on, transformations to be performed, and the output data type. Therefore, performing a subtraction between two features in Aligned will not perform the subtraction directly, but rather describe a derived feature with the expected output, the features needed to compute the value, and lastly the subtraction transformation with the related feature ordering as shown in figure (3.2).

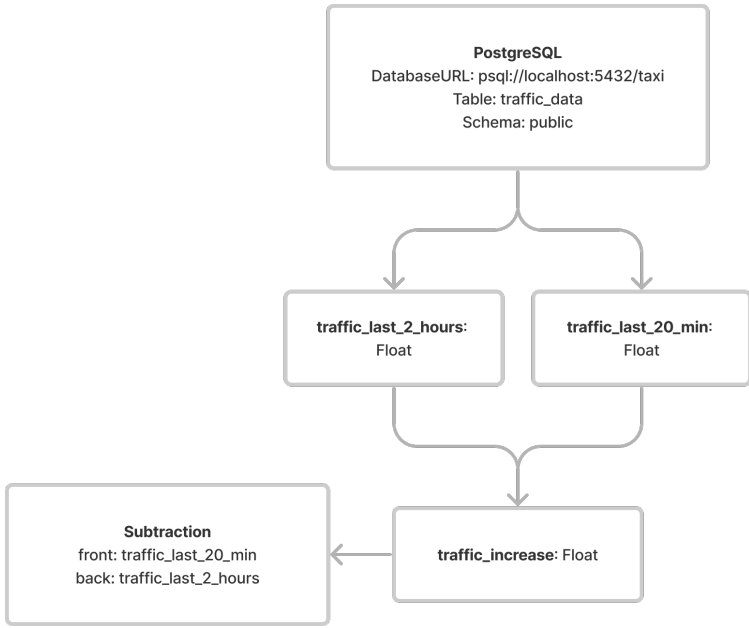

Figure 3.2: The derived feature graph of a subtraction feature

3.2.3 Transformation

This also means that all the transformations between features, can be represented as a JSON object. For instance, a subtraction transformation, can be represented as the two names of the columns that should be subtracted. Representing transformations in such a format is core to the solution, as it makes it possible to derive the whole data lineage, without relying on any specific technology.

Therefore, computations can be optimised and increases developer flexibility, as developers will not be locked into any processing engines. Enabling to switch from Pandas, to Polars, or to Spark in seconds.

Aggregated Feature

Similar to derived features, aggregated features involve transformations on data within a time slice. For example, computing the average passenger count over the past 20 minutes for a taxi company represents an aggregated feature.

Here will the usage of an event timestamp be important for aggregated time windows, as Aligned will setup the needed logic to only compute over the time window of interest. Which is shown in the taxi example in section (4.1.1).

3.2.4 Model

To achieve the intended objectives of finding data dependencies in our ML systems, it is necessary to represent the behavior of our model. To accomplish this, Aligned enables the representation of the data flow associated with a model.

At its most fundamental level, this entails describing the input and output features of the model. However, a model contains different types of output data. Therefore, it is also possible to describe additional information related to the model.

Label

An ML model will produce some kind of output. Therefore, Aligned makes it possible to describe the output, or our $\hat{y}$.

However, our $\hat{y}$ will differ based on the ML use-case. Therefore, Aligned makes it possible to distinguish between classification, regression, and unsupervised outputs. However, classification and regression have received the most support.

Prediction Source

In many cases, it is valuable to store the predictions made by our model. Consequently, Aligned allows for the definition of a source where historical predictions

can be accessed. Similarly, real-time predictions can be obtained through streaming.

Moreover, a prediction may include more than just the predicted value itself. For instance, metadata such as the model version and entity IDs can be stored alongside the predictions. Additionally, metrics like prediction certainty and class probabilities may also be of interest to store.

Aligned provides the capability to include such data within each prediction, enabling various interesting functionalities which will be elaborated upon in the subsequent section (3.3.8).

Ground Truth

However, having access to predictions and their associated metadata alone is insufficient. To effectively evaluate the performance of our models, knowledge of the locations and timeliness of the ground truth values is necessary.

To a certain extent, Aligned can identify where historical ground truth values can be retrieved based on the indirect dependency graph on a feature view. Consequently, this works well when creating datasets, as we can specify the entities for which data sets should be generated. However, the same approach cannot be applied when observing real-time ground truth. In such scenarios, it is imperative to ascertain when a value represents a valid ground truth. For instance we may want to trigger a ground truth event when a categorical value is set to a subset of values.

To address this, Aligned enables the registration of streams where ground truth values will be received. This can be accomplished by specifying the condition and the sink for the ground truth or by describing a stream source where the values will be stored. Consequently, the choice is available to either let Aligned handle the ground truth condition or manage it manually outside the scope of Aligned.

3.2.5 Metadata Storage

The primary objective of this solution is to represent the data flow and data lineage in a high-level system, without delving into the specifics of the internal implementation. Consequently, it was crucial to enable storage of all this metadata in a language-independent format.

This requirement prevent the use of storage formats such as pickle or dill, which are commonly used in Python.

Instead, the solution is designed to represent the data in any desired format. Therefore, enabling compact storage using protocol buffers, or a dynamic PostgreSQL storage with added data governance. Making it possible to select the representation that makes the most sense for the use-case. However, by default,

Aligned supports the JSON data representation as the initial data storage format, mainly due to its human-readability, widespread use, and fast iteration speed.

3.3 Implementation Compiler

Up to this point, we have only succeeded in describing our system in a decodable format. Although this contributes to thorough documentation, it holds little value if we can not derive functionality from our documentation. In fact, this discrepancy can lead to the system quickly drifting out of sync, thereby diminishing the utility of Aligned.

The true value of this approach lies in its potential to set up common components needed for a ML system, see section (2). This constitutes the final component that unifies all the elements, transforming our documentation into an actionable, value-adding resource.

3.3.1 Data Catalog

Arguably, one of the most essential components of this solution is a data catalog. Given the vast amount of information compiled about our system, it is reasonable to provide a user-friendly interface for viewing this information.

The data catalog allows users to browse through all features and models, and to view the current state of the system. Consequently, it provides a comprehensive understanding of all our existing data, where it is stored, and how it can be used.

Figure (3.3) illustrates several features for the *taxi_vendor* feature view, which is associated with the taxi model described in section (4.1.1). This provides clarity on where the historical data will be stored, where real-time values will be processed from, and how the different features depend on each other.

A similar page exists for the *taxi* model itself, as shown in figure (3.4). This page displays the features used, the required entities, and the data lineage for all features. It also enables the view of features from a low latency online source, if available, and real-time performance metrics if ground truth values are monitored, as discussed in section (3.3.7).

3.3.2 Data Lineage

The provision of data lineage enhances our ability to understand and debug our data logic. Despite the utility of the documentation feature, there are additional reasons why data lineage is crucial for the functionality of Aligned.

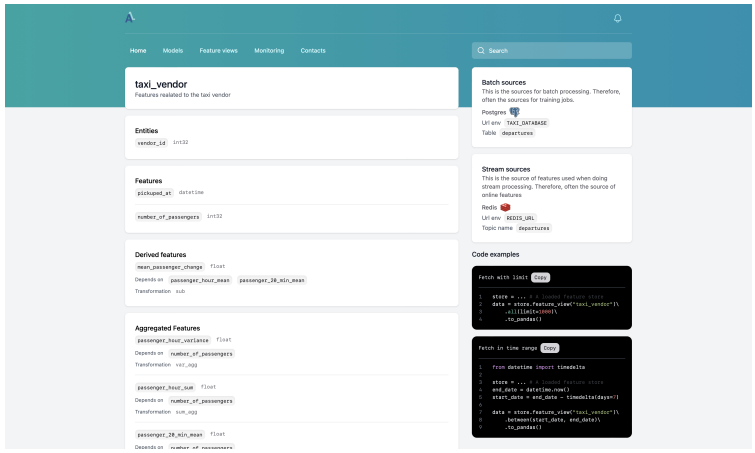

Figure 3.3: Feature documentation

Value gain

Feast, an open-source feature store, lacks data lineage features. Consequently, end-users are required to know the data lineage themselves, which can lead to common errors. The end-users might not comprehend the entire data lineage and data dependencies, which can cause issues when running feature transformations. For instance, if a feature assumes it has access to another feature that isn't loaded into memory, this could lead to run-time errors and crashes.

To address this problem, Aligned has made it possible to define data dependencies and lineage implicitly using Python's descriptor[1] feature.

Furthermore, because each transformation returns a new type, and Python's dunder methods allow for a custom implementation, we can create a complex graph of dependencies that is largely transparent to the end-user. As a result, users can focus on business logic in an intuitive manner, while Aligned handles the compilation of dependency logic. Leading to a data lineage graph similar to figure (3.5).

Therefore, we get the same functionality as DBT's ref feature, but with a higher level of detail and flexibility. This because we can both get the lineage for individual features, but also different sources of data, and for different types of storage. For instance could some features be stored in S3 while others in a DWH like Redshift, and lastly some in a production db like PostgreSQL.

However, there are even bigger advantages of this fine grained data depen-

[1]Python descriptor - `https://docs.python.org/3/howto/descriptor.html`

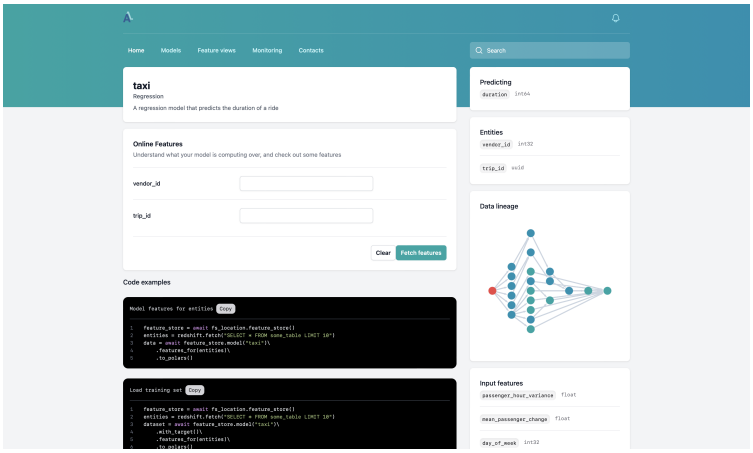

Figure 3.4: Documentation of our ML models

dency graph.

Feature Optimisation

A significant advantage of such a system lies in its potential to reduce the required computational power. By employing our dependency graph, we can decrease the necessary compute.

For instance, if we request the feature *day_of_week*, we only need to load the *picked_up_at* feature. This avoids the need to load extensive coordinate data or to compute an unrelated Euclidean distance.

Furthermore, Aligned can remove features from the computation graph at a system level, since Aligned knows which features to use in our models. Therefore, understanding which features are dead code.

This feature optimisation becomes particularly apparent in columnar databases, where we only select the absolute minimum data needed. This leads to reduced network traffic and faster read times.

Technology agnostic

Another significant advantage of such a system is its adaptability to multiple technologies in a matter of seconds. Since we do not define the specific process used for feature handling, but rather only establish how features relate to each other, the system maintains flexibility. This flexibility allows seamless transi-

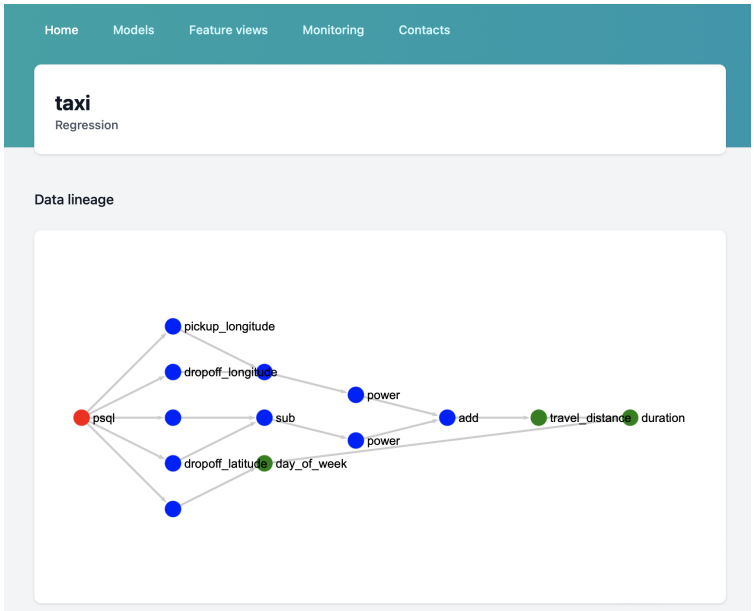

Figure 3.5: Data lineage for a taxi model

tioning between technologies such as Pandas, Polars, Spark, or SQL depending on user needs. Thus, it guards against technological lock-in, providing greater user flexibility and reducing development costs as a business grows and older technologies fail to handle the new requirements.

3.3.3 Feature Store

The feature store, providing a centralized source of truth for our features, has emerged as an essential requirement in ML systems. Given the detailed description of our features, their location, and their interrelations, this is an area where Aligned can provide significant value. Consequently, the feature store compiler can efficiently manage the features.

Historical features

Once the feature store has read all the metadata, it will, by default, load features from the batch source. This capability enables the loading of historical features.

For instance, the feature store can execute a query to load features for a model,

alongside its target feature, for entities with a valid ground truth in the preceding 30 days. The feature store then proceeds to load data from all data sources, appropriately combining them and computing additional features as needed.

The feature store also ensures point-in-time correctness of the features, based on the provided event timestamp. Moreover, it can create training and test splits. While this might seem unremarkable, having predefined feature behaviors enables us to predict if our data will be independently and identically distributed (i.i.d.). This facilitates the automated setup of training and test splits based on timestamps, effectively preventing time leakage.

Our proposed solution mitigates the potential for errors by deviating from the API choice seen in scikit-learn's split function. Instead of returning four separate variables, which could lead to confusion due to two valid representations ($X_{train}, y_{train}, X_{test}, y_{test} = split$, and $X_{train}, X_{test}, y_{train}, y_{test} = split$), our solution returns a single variable containing both the train and test sets as properties. It further groups different columns based on their intended usage such as features, label, event timestamp, and entity columns.

Inference features

However, loading historical features can occasionally be a slow process due to the underlying storage engine, which may have been optimized for different types of read patterns. Additionally, we may load more data for training than for inference, such as the ground truth value.

This is why the feature store can load data from an online source. Commonly, this would be a key-value store like Redis or Apache Cassandra. Consequently, the feature store will offer a similar API to the historical features method, but end users have the flexibility to define the feature logic's location, making it adaptable and standardized.

3.3.4 Data Testing

Validating and testing that our data conforms to our assumptions has shown to be an important step, as described in section (2.1.8). Furthermore, by declaring features using higher level data types as mentioned in section (3.2.2), enables us to generate data quality tests. Therefore, validating that our data conforms to the expected schema, and semantic in a column wise manor while keeping the underlying technology flexible.

3.3.5 Feature Server

Establishing an online source can necessitate additional infrastructure work, such as setting up a Redis cluster. Developers might want to avoid installing extra

dependencies on a server that only handles predictions. Consequently, they might prefer not having the feature store in memory, but rather as a separate service. This could be due to the usage of a programming language such as Rust, which may not yet support the feature store.

This is where the feature server proves beneficial. We can specify our requirement for a server code that functions as an online source, and Aligned will facilitate the structuring of all API calls, generate an OpenAPI schema for documentation, and handle reading from the online source. Therefore, a server can be set up with only 3 lines of code.

3.3.6 Stream processing

Stream processing is closely related to the feature server. In this scenario, Aligned will spin up a process that listens to all stream sources, computes the derived and aggregated features, and stores them in the online source. This separates the concerns of reading and writing, effectively populating the online source with features.

Aligned provides a straightforward streaming implementation that leverages the Polars framework. This solution maintains a low footprint while offering considerable processing power. However, in theory, Aligned can compile a PySpark, Spark, or Flink application with identical logic, providing users the flexibility to select the technology that best suits their needs at any given time.

3.3.7 Model Monitoring

Monitoring performance for online models can be difficult due to a number of reasons, one of which is that the ground truth of our predictions only becomes available in the future. In the worst-case scenario, it may take months or even years before we receive the real outcome.

This implies that an ML system requires a plan for long-term storage that can be accessed whenever needed. Fortunately, at a high level, there are two main requirements to set up model monitoring.

Ground truth source

We need a source that defines what the ground truth is. From my research, it appears that this often originates from the same source as our labels used in the training set. However, in some cases, a set of conditions may be necessary to constitute a ground truth event.

Presenting this data in an event-based representation can simplify reasoning if it's stored in a database setting. Consequently, being able to tune into a *ground truth stream* can be quite valuable.

Prediction source

The second requirement is the predictions themselves. Given that the predictions are likely generated before the ground truth, they need to be stored in a long-term storage solution, such as a persistent database. Thankfully if the developer have define this storage in the model interface will Aligned know where to fetch them.

Entity alignments

In other more mathematical terms. The entity ids for each prediction needs to be fully included in the ground truth's entity ids.

Being able to connect predictions to ground truth events is essential for model monitoring. Therefore, some kind of entity ID is needed. Since the ground truth will arrive later than the predictions, it is crucial that the predictions can be retrieved using the entities associated with the ground truth event. An SQL query similar to *SELECT * FROM ground_truth_event LEFT JOIN predictions ON ground_truth_event.entity_id = predictions.entity_id* could be used. In mathematical terms, the entity IDs for each prediction need to be fully included in the ground truth's entity IDs.

Problem type

The model's evaluation also depends on the type of ML problem since classification and regression tasks differ in how their data behaves. Luckily, the logic can be deduced from the type of problem.

Classification monitoring

For a classification model, four different states can occur: True Positive, True Negative, False Positive, or False Negative. Counting these four states for each class makes it possible to aggregate metrics like accuracy, precision, recall, and F1. Furthermore, by monitoring the count for each class combination, users can identify classes that may require more training data.

Regression

For regression models, the core measure to aggregate is the distance to the real value, $v_{ground\ truth} - v_{predicted}$. This metric can be used to compute others such as mean square difference, absolute difference, and square root mean square difference.

API

The API to define such logic is critical. The goal of this project is to make it simple yet clear. Thankfully, only 5 lines of code are needed to set up such a feature. As described in section (3.3.7), a source for predictions is needed, which is shown on line 7 in figure (3.6), while the ground truth source is defined on lines 19-22.

```
1  class TitanicModel(Model):
2      passenger = TitanicPassenger()
3
4      metadata = Model.metadata_with(
5          name="titanic",
6          description="...",
7          predictions_source=titanic_source,
8          features=[
9              passenger.constant_filled_age,
10             passenger.is_male,
11             passenger.is_mr,
12             passenger.has_siblings
13         ],
14     )
15
16     # A condition be needed, and where to sink the ground truth
17     # Since the ground truth is a part of the feature view
18     survived = passenger.survived.as_target()\
19         .send_ground_truth_event(
20             when=passenger.survived.is_not_null(),
21             sink_to=redis.stream(topic="passenger_ground_truth"),
22     )
```

Figure 3.6: A model definition of a Titanic model, with model monitoring setup

Since the *TitanicPassenger* feature view includes a stream source, a stream processor will listen to features that have a *survived* feature defined, and send them to the stream topic *passenger_ground_truth*. It is also possible to define ground truth events that do not listen to the original feature view, using *listen_to_ground_truth_event*. However, this requires setting up custom condition logic and the expectation that all events arriving at that topic will be valid.

Aligned has its own worker logic that listens to the defined ground truth topics and executes the necessary logic to generate metrics accessible to the rest of the system.

Infrastructure

Supporting such a feature necessitates a time-series database to compute performance over a time window. Tools such as Prometheus can measure and aggregate metrics efficiently.

Prometheus is a service that collects metrics over time, providing an API to fetch and aggregate metrics. Such metrics could include CPU and memory usage, or more use-case-related metrics like the number of true positives or distance to ground truth. Therefore, Prometheus stores our metrics in a format similar to the following: *number_of_true_positives 30*.

Being able to select only a subset of metrics is vital. Prometheus provides this feature through labels, leading to a format similar to *number_of_true_positives*{*model='MNIST', ground_truth=3*} *30*.

This integrates well into the existing architecture that uses Kubernetes. Prometheus has good integration with Kubernetes, making it relatively straightforward to set up while enabling advanced queries for real-time performance. However, this necessitates setting up the metrics correctly. Since we need to differentiate between different models and model versions.

3.3.8 Data Centric AI

The introduction of data centric AI have been an important influence on this project. Therefore, some data centric features have been added as a result.

Semi-supervised learning

Semi-supervised learning can fortify models by estimating labels. Consequently, semi-supervised logic can be applied by defining labels based on transformations. A plausible solution is filling in missing ground truth with a heuristic, as demonstrated to some extent by Snorkel.ai.

Self-supervised learning

Establishing self-supervised learning pipelines simply involves structuring your transformations and labels appropriately within the data flow. For instance, data can be generated for a 'predict the next word' use-case, where the last word of a sentence is removed and used as a label.

Active Learning

Working with annotated data can be challenging, but active learning can optimize the utilization of our annotations. Consequently, we have introduced the capability to generate active learning datasets.

3.3.9 Large Language Models

The emergence of ChatGPT has led to a surge in the adoption of Large Language Models (LLMs). As a response, we have incorporated support for LLM embeddings.

Embeddings

Embeddings enable a novel representation of semantics. As such, basic embedding support can be highly influential. Aligned, thus, supports three of the most widely used frameworks for sentence embeddings: OpenAI, HuggingFace, and Gensim.

Offering support for these three distinct frameworks enables Aligned to accommodate a broad spectrum of use-cases and requirements. For instance, OpenAI may appeal to users seeking the most potent models, despite the associated cost. On the other hand, a different user might prefer an open-source LLM, even if it necessitates managing compute costs. Lastly, Gensim provides a low-compute alternative, albeit without the most potent results. While Gensim doesn't neatly fit into the LLM category, the mean of different word vectors can emulate a sentence embedding.

Vector Databases

The computation of sentence embeddings can produce valuable insights and enhance the interface. However, integrating support for vector databases aligns well with the existing support for the feature server and low latency feature storage. Hence, the automation of much of the work required to set up vector databases can be achieved by selecting the correct storage format in our online feature storage. Redis serves as an example of a technology where the feature storage mirrors that used for vector search. The only additional requirement is the setup of a vector index. This led to the natural progression of defining vector indexes, with Aligned managing the ancillary metadata and commands necessary for their establishment.

Chapter 4

Experiments and Results

This chapter describes the experiment that evaluate the proposed solution and presents the obtained results.

4.1 Experimental Plan

To ensure that the proposed solution delivers its intended value, an experiment is conducted. A range of machine learning (ML) use-cases are implemented using the proposed solution, thereby simulating a business environment with multiple deployed models, and an environment where the technical debt described in section (2.2) begins to accrue.

The selection of different types of use-cases is crucial as the solution needs to be versatile enough to accommodate a broad array of ML products. Consequently, the solution is analyzed in light of the technical debts outlined in section (2.2), thus enabling a more objective appraisal of the design choices.

However, this is only one aspect of a solution. Therefore, a comparative study is set up between the presented solution and an equivalent one using more conventional technologies. This comparison leads to a series of interviews to gather impressions and concerns about the different solutions.

4.1.1 Use-cases

A comprehensive presentation of the use-cases, along with reasons why they constitute a good representation of real-world examples, is given. All use-cases can be found on Github at MatsMoll/aligned-example[1].

[1] Aligned Examples - `https://github.com/MatsMoll/aligned-example`

New York Taxi

The New York taxi dataset[2] is a well-known dataset, making it a good use-case for testing. Therefore, it simplifies the process of testing and comparing existing solutions. Furthermore, the dataset can naturally simulate a database environment where the labels and inputs are stored in different tables.

Moreover, this use-case illustrates how we may want to represent the same data from different perspectives, as we aim to select the newest features related to a trip id. Simultaneously, we are computing aggregate features over a time interval related to a vendor id.

Thus, the combination of multiple database tables and time window aggregations render it complex yet relevant for real-world applications.

Titanic

The Titanic dataset is often used to learn basic data science concepts, as the dataset provides opportunities for one-hot encoding, filling in missing values, and implementing other basic ML methods. Therefore, the Titanic dataset was included to demonstrate how these methods could be implemented.

Credit Scoring

The finance sector is known for utilizing real-time ML because of the need for low-latency inference, allowing for faster fraud detection and increased revenue. This explains why Feast uses a fraud detection model as one of their examples[3]. The same use-case is presented, but instead of using a BigQuery[4] database, the example employs a data lake approach with multiple parquet files.

Furthermore, having access to such an example makes it possible to see how well-known technologies solve the same use-case and where improvements can be made. Therefore, it presents an interesting scenario and facilitates easier comparisons.

MNIST

MNIST is a commonly used image dataset. Therefore, setting up MNIST as a use-case will clarify how image tasks can be supported, providing insight into the flexibility of the solution, and ensuring it is generalizable. This use-case will demonstrate how image data is often stored with a reference to a remote file

[2]New York Taxi Dataset - `https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page`

[3]Feast Fraud Example - `https://github.com/feast-dev/feast-gcp-fraud-tutorial/blob/main/notebooks/Fraud_Detection_Tutorial.ipynb`

[4]BigQuery - `https://cloud.google.com/bigquery/`

and present how it could be possible to describe and extract the image data in Aligned.

Documentation Chat Bot

Keeping documentation up-to-date can be challenging. Therefore, a large language model (LLM) using in-context learning to explain how to use the proposed solution was set up. In this model, all documentation and code are indexed using an LLM embedding and stored in a vector database. Consequently, it is possible to extract relevant information needed to answer a user's question. This information would then be inserted into the context along with the user's question, and then sent to an LLM to generate a response. Therefore, demonstrating how LLMs can be supported using the proposed solution. The described bot will be available in Aligned's documentation[5].

Categorization

Each use-case is categorized into different categories. They are classified based into how the ML problem behave, as shown in table (4.1). Here will it present the type of ML problem, the behavior of the label, and which data type the predicted label is. Furthermore, a second classification is done based on the data format, and the data behavior of the ML use-cases, as shown in table (4.2). Here are the data source type shown, which slowly changing dimension is used, as talked about in section (2.1.8), and which type of data format the model uses.

Table 4.1: Use case ML Problem

Use-case	ML Problem	Label Behaviour	Label Type
New York Taxi	Regression	Natural	Int
Titanic	Classification	Natural	Bool
Credit Scoring	Classification	Annotated	Bool
MNIST	Classification	Annotated	Int
Docs Chat Bot	Recommendation	No labels	No labels

4.1.2 Interview plan

A semi-structured interview is conducted, during which two different implementations of the same ML training pipeline are presented to the interviewee. Questions are asked to ascertain how quickly the interviewee comprehends the system-wide logic and which part of the solution could be more intuitive.

[5]Aligned Documentation - `https://www.aligned.codes`

Table 4.2: Use case Data Source

Use-case	Source	SCD	Input format
New York Taxi	PostgreSQL	1	Tabular
Titanic	CSV	1	Tabular
Credit Scoring	Multiple Parquet	2	Tabular
MNIST	CSV	1	Image
Docs Chat Bot	CSV	2	Text

Consequently, it becomes possible to evaluate if the proposed solution effectively addresses the intended problems.

4.2 Results

The results are derived from six different interviews, where all participants were either working or studying in the field of AI. See appendix (5.8).

4.2.1 Taxi Training Pipeline

Two different training pipelines were presented to a set of intervieews. First, a pipeline implemented in SQL, with point-in-time streaming aggregations, schema validation, and training a gradient boosted model presented. Thereafter, the same pipeline was presented using the Aligned solution. The first initial response was that all participants were impressed by the minimalistic code necessary to set up a training pipeline.

SQL

Handling SQL can be challenging given the multitude of styles and it being a distinct skill to learn. Participants remarked that the proposed solution provides a useful way to translate SQL logic into Python, thereby reducing the need for advanced SQL knowledge. Furthermore, one participant appreciated this aspect as it mitigated disputes over SQL code structuring, thus eliminating more subjective arguments when working in a team.

Pipelines

Managing a wide range of pipelines can be tricky. Therefore, the participants mentioned that the proposed solution was an improvement, as minor sub pipelines can be described, and then combine them into larger ones. Furthermore, the

developer can add pipeline steps as they wish. Like validation, training validation data-set splits.

However, this also had their downside, as the sub-pipelines often would be split into separate files. Making it harder to lookup the code that described the full pipeline.

Data Lineage

Since it could be harder to reason about the pipelines, was the graphical UI shown with the data lineage graph shown. The data lineage helped to make the pipelines clearer, as the full pipeline could be show in a graphics, rather then code.

Data catalog

Everyone that was interviewed thought the data catalog provided useful information. Mainly the data lineage graphs as the biggest advantage. However, they also enjoyed the possibility to search and find existing features. Furthermore, the interviewees enjoyed the models real-time performance metrics.

Abstraction

The participants felt the solution provided a wide range of functionality. However, there were some friction.

Mainly, the participants that study AI struggled to understand the abstraction concepts. While the working interviewees found it to be more intuitive. The AI students saw that the solution lead to less code, but struggled to follow the different concepts, and how it related to AI. One such confusion was the Model concept. They often though it would actually run the model. Therefore, the intended solution to only managing the data around the model was miscommunicate.

Furthermore, some participants was afraid they would loose flexibility in the system, and get locked into the new solution. However, it was explained that this would not be the case, as the developers could use only the data loading features, and not any processing, validation, or other system. Therefore, making the developer control the amount of dependency on the system.

4.2.2 Credit Scoring Pipeline

Late in the project was a comparison against Feast[6] was setup, which can be seen in section (4.1.1). Feast is the open-source feature store library the most starts

[6] Feast - `https://feast.dev`

on Github. Therefore, it was seen as a good comparison to test the feature store component. However, because of the late example was less information on this pipeline collected. However, the interviewees that was asked about this pipeline provided useful insights. The interviewees mentioned that the proposed solution was a huge leap, and that *You have nothing to envy to Feast, on the contrary.* However, there was raised some concern about the nature to manually type out the data schema of files and tables.

Chapter 5

Evaluation and Conclusion

5.1 Evaluation

The following chapter mentions the technical debts described in section (2.2) and how the proposed solution addresses them. Therefore, this section will be an analytical analysis of the core design choices described in section (3), and how it effects our technical debt.

5.1.1 Entanglement

The proposed solution facilitates a mechanism to declare that the output of one model will serve as the input for another. While this functionality alone doesn't resolve the CACE principle: Change Anything Changes Everything, it illuminates potential CACE scenarios. Thus, the proposed solution enables monitoring for each output variable, by employing data drift methods described in chapter (2).

5.1.2 Correction Cascade

Recognizing when a new model enhances an use-case's performance can be challenging, particularly in situations involving model chains. To mitigate this, the proposed solution can alert system developers of expanding model chains by using the data lineage as showing in section (3.3.2), offering suggestions on eliminating such dependencies.

5.1.3 Undeclared Consumers

As the proposed solution outlines various data sources, transformations, and data lineage for features and model outputs, it helps us understand who consumes and

writes data in the system. It clarifies access requirements for different information and, to an extent, controls data writing permissions. This results in enhanced system visibility. Additionally, the data catalog described in section (3.3.1) offers a graphical interface to illustrate this more intuitively.

5.1.4 Underutilized Data Dependencies

The utilization of data can fluctuate over time. Accordingly, the data lineage functionality can identify when features are in use or deprecated, thereby alerting users about feature usage. Moreover, the stream processing setup through the implementation compiler described in section (3.3) can detect features with primarily missing values, notifying about legacy features as per section (2.2).

We could also consider integrating SHAPLY values to examine bundled features. Additionally, we could monitor if a new model outperforms an older one, thereby checking for ϵ-features, and determining whether the addition of features is worthwhile.

5.1.5 Static Analysis of Data Dependencies

Owing to the system's awareness of the features used in a model, as indicated in section (3.3.2), the data lineage information allows the computational graph optimization across different systems. This enables the removal of redundant features, reducing the computational load of the ML system overall. Furthermore, we can monitor different systems and analyze which data dependencies are safe to remove based on system usage.

5.1.6 Glue Code

The proposed solution offers a unified interface for all types of data sources, as shown in section (3.3.3). As a result, end-users are relieved from the need to maintain glue code, and a simple API is set up, allowing them to concentrate more on the ML logic rather than on system "plumbing."

5.1.7 Pipeline Jungles

The proposed solution conscientiously supports multiple diverse data sources simultaneously. Additionally, by providing a standard method to manage different pipelines, it simplifies testing of the entire pipeline. Therefore, incorporating new signals becomes no different from adding any other source.

5.1.8 Dead Experimental Codepaths

The proposed solution includes a data dependency graph linking every input, transformation, and output, facilitating the emergence of new solutions. Specifically, it allows for the graph's analysis, identifying redundant features and functionalities, thereby enabling the automatic removal of dead code.

5.1.9 Plain-Old-Data Type Smell

Data engineering often employs plain-old-data types such as integers, floats, or strings, obscuring valuable information. Hence, the proposed solution enables the definition of higher-level data types, as presented in section (3.2.2). Some examples include coordinates, URLs, or emails. This allows format validation, for instance, verifying the format of a URL, while also providing standard transformations like "extract the domain from the URL."

This approach facilitates the identification of personal identifiable information (PII), as the system can notify when such information could potentially emerge, thereby reducing the probability of data leakage of sensitive information.

5.1.10 Multiple-Language Smell

The maintenance of multiple languages can lead to systems that are difficult to manage due to the challenge of finding developers proficient in several programming languages. Therefore, striving to do as much as possible in one language is desirable. Python has become the most common language for ML applications, so defining SQL queries and high-level system architecture in one language can reduce the cognitive load for developers, making it easier to onboard. Consequently, the implementation compiler section (3.3) allows for code development in one unified language, while still benefiting from different languages when necessary.

5.1.11 Prediction Bias

Since evaluating model performance can be tricky, and ML practitioners often have limited trust in local evaluation metrics, as stated in section (2.1.16), addressing prediction biases is critical. To that end, the proposed solution makes it easy to set up real-time evaluation metrics, as shown in section (3.3.7).

5.1.12 Data Testing Debt

While testing code and models is a recognized practice, testing data isn't as common but is equally important to catch incorrect assumptions earlier, discover

anomalous data, and identify malicious data. The proposed solution enables feature validation, as detailed in section (3.3.4), by enforcing different data types and inspecting text lengths, upper bounds, lower bounds, missing values, etc. Furthermore, knowing where to fetch predictions and ground truth values allows for leveraging data critical techniques such as creating regression test sets.

5.1.13 Reproducibility Debt

Reproducibility is a crucial component in ML. Therefore, elements like model registries and version control are essential. However, the same requirements apply to data, necessitating data set versioning. However, the data set versioning should be based on point-in-time correct data or log-and-wait features. Otherwise, the features won't represent the real-time-traveled data, potentially leading to poorer models in production.

Hence, the proposed solution enforces point-in-time correct data joins using event timestamps, as outlined in section (3.2.2), while also enabling the setup of log-and-wait features through the implementation compiler, as detailed in section (3.3).

5.1.14 Interviews

The following subsection will evaluate the interviews and the results described in section (4.2).

Abstractions

While data abstraction was well-understood by interviewees working in AI, it was less clear for the students. A possible solution involves renaming different concepts and providing a more explicit explanation of the core problem space. However, the proposed solution was seen as an improvement against existing well known solutions.

Find transitive data dependencies

Finding transitive data dependencies is what Aligned is designed for, and the interviewees found that the data catalog helped presenting data dependencies in a clear manor. Therefore, such problems should not be a problem to the same extent as before. We can easily figure out that model m_A's output is sent further into another model, or that we have a feature that goes through 10 steps of processing before being used in a model. Because of this will it also be possible to remove features from our ML system that is never used.

Measure the effect of a change

Aligned makes it possible to measure the real-time performance metric for a model, something the interviewees appreciated. Furthermore, since we have a mapped out dependency graph, will it be possible to monitor and find signals that can be unstable.

5.2 Discussion

This section discusses various design choices and their implications for the end result.

5.2.1 Transformation system

One crucial choice has been the representation of transformations. Instead of representing transformations as pure code, they are depicted as unique identifiable names accompanied by the configuration required to complete them. For instance, an integer clip transformation would represent the feature to clip and the upper and lower bounds for clipping.

This approach eliminates any dependencies on the processing engine, thereby enhancing system flexibility to adapt to technological changes. However, the language-independent representation of a transformation increases the work needed to add new transformation types. This is partially mitigated by support for User Defined Functions (UDFs), although UDFs can limit user flexibility as they rely on Python code and require a Python runtime.

5.2.2 Feature versioning

Implementing version control for features is as essential in large-scale ML systems as it is in software design and DevOps. By describing the system at any point in time using solutions like Git, we can version control new features and archive feature definitions as versioned artifacts and store them using AWS S3 or another blog storage.

5.2.3 Choice to not define an online source

Some feature store solutions have decided to define the online source in the feature definitions. This was initially what the proposed solution also did. However, after some more thought was this reverted and the online source was deleted from the definition.

Therefore, it is the developers choice to select the sources that makes the most sense for their use-case. As a result, the system manages to get the same behavior as an online source.

Furthermore, such a design also makes it possible to test out and use multiple sources if needed. Such a use-case could be interesting where some features have a lower latency requirement. Something Robinhood have mentioned they needed, where they combined AWS Dynamo DB, and Redis to balance latency and cost.

5.2.4 Debugging

Some code will eventually contain a bug, or have an unexpected effect. Therefore, having an efficient way to debug the system will be important.

However, the solution focus on describing the system, and deriving the implementation details from a high level description. Therefore, it means that the user have less control over the code, which can make it harder for the developer to fix bugs. Rather, the developer would need to add pull requests to the library code base. However, this can potentially lead to a higher quality code base, as it will be more thoroughly tested.

Thankfully, the solution have added the functionality to describe the intended logic that will run. Therefore, it is possible to inspect, and find where errors can occur faster, while understanding what the system will do.

5.2.5 Aggregation features

Making the decision to support aggregation features was an important, as they can be expensive and hard to compute. Because of the difficulty of aggregations is this a likely source of data debt. It is easy to compute aggregations over incorrect data windows, leading to poor ML performance. Furthermore, such a decision have been supported by Airbnb, where they have mentioned that aggregations have been an important, yet challenging feature.

Therefore, supporting aggregations was essential to create a good foundation, even though the underlying computation of the aggregations have not been the main focus. However, such a choice have lead to the decision to put other model related features on hold, which would be more unique. But such features would be

5.2.6 Testing new features

One of the questions that could identify technical debt in Sculley et al. [2015], was how quickly we can move and add new functionality. And the same thing matters when it comes to data. How quickly can we move and test out new features and data. Thankfully, the design choice to support multiple different data sources,

and then combine them was intentional. Therefore, making it possible to test out features in one database before notifying a data engineer about loading the same data into a more proper data warehouse. However, this can have a downside as bad practices can be used, as the incorrect infrastructure is used in an unintended way, leading to poor performance.

5.3 Contributions

This master thesis have lead to the following contributions.

5.3.1 Talk

A presentation on the problems, findings, and proposed solution was held for the MLOps community in Oslo, where the solution was well received and sparked numerous interesting discussions.

5.3.2 GitHub

Furthermore, the proposed solution is accessible as a Python package called *aligned*, and accessible as a open-source project. Furthermore, the examples analysed in section (4.2) are also available in it's own repository called *aligned-examples*.

5.4 Conclusion

This master thesis has explored the concept of technical debts in machine learning (ML) systems, highlighting their potential to undermine the effectiveness of ML products. The research has explained how detailed information concerning data sources, feature processing, and models-collectively termed 'data lineage'-can enhance the transparency of data flow within a system, thereby facilitating the identification of potential errors.

In addition, this study has introduced and examined the potential of 'Aligned', an approach to collect data lineage information and leverages associated metadata to mitigate common data errors through a metadata compiler and an implementation compiler. This exploration has extended to demonstrating how such metadata can be deployed to setup frequently-used ML components, such as feature stores, model monitoring, and data catalogues, thereby augmenting the overall quality of ML systems and enhancing their flexibility. Therefore, mitigating technical data debt.

Through an in-depth analysis and demonstration of these elements, this thesis underscores the crucial role of data lineage and behavioral descriptions through metadata to improve the robustness and versatility of ML systems, ultimately fostering better product outcomes. The insights drawn from this research could serve as a foundation for future studies aiming to further optimize and enhance deployed ML systems.

5.5 Future Work

Even though there have been a lot of work completed in this master project, have this only been the needed foundation. Therefore, the main focus have been to collect a wide range of metadata, mapping out data dependencies to find potential debt.

However, even though the solution can find debt, does it not mean it can act upon the information. Therefore, future work would be focused on acting on such information.

5.5.1 Unstable Data Dependencies

Currently there are no data drift detection solution setup. However, setting up such a functionality should be fairly easy. The only requirement is that a reference to a data set exists. Therefore, making it possible to derive the expected distribution and compare them to the served features, and predicted outputs. Furthermore, there are technologies such as Evidently AI that can be leveraged to implement more advanced data drift techniques. Therefore, a more generic interface would be needed to make it possible to potentially combine, or leverage multiple existing solutions.

5.5.2 Prediction Bias

Evaluating model performance can be hard, and even though a model with 99% accuracy may sounds like a well function model. Could it still be that a model predicting the majority class, or the average can perform as well, or even better. Therefore, the proposed solution can make it possible to catch such prediction biases earlier, as an alerting system against baseline models can be setup. To setup such a feature, would only one artefact be needed. The training data set. Therefore, if the solution adds a way to link data sets to a model, can basic strategies be setup to compare against a baseline model. However, it is also possible to compare different models against each other in the cases where the baseline model is a too low baseline.

Furthermore, increasing the support to monitor our model performance based on a subgroup could be interesting to explore. Hence, potentially improving the performance for minority groups.

5.5.3 Correction Cascade

Knowing if a model performs better then a simple baseline is one thing. However, testing multiple models in production at the same time is another thing. Therefore, the proposed solution can help setup A/B testing, Canary deployment, or even bandits section (2.1.5) to determend which models work best. This is made possible since we have access to both the y and $\hat{y}$, and can therefore keep track of performance per model. However, either a new service, or an improved implementation of the monitoring system would be needed.

5.5.4 Reproducibility Debt

Reprocudibility is essential in ML systems, and as mentioned in section (5.1) the proposed solution have implemented point-in-time joins. However, implementing another solution such as log and wait features would be an interesting solution. Therefore, making it possible to time travel in data, with a simpler data architecture.

5.5.5 Prototype Smell

The current solution have mainly been developed for technologies that do not rely on distributed computing. Therefore, it has some prototype smell, as common processing engines like Spark, and Flink is not supported. Furthermore, common streaming data sources like Kafka has not been added either. Therefore, adding support for such technologies would be needed to make it usable in a wider range of problems.

5.5.6 Data Testing

Testing our ML models is a complicated topic, with many nuances. However, since the proposed solution knows where where both our input, ground truths, and predictions can be stored, can it improve testing. Therefore, it enables us to create regression test sets, and integration test sets with ease.

5.5.7 Model evaluation

Setting up automated evaluation for our models was added in this project. However, there are still improvements that could be made. One such improvement

would be to add real-time evaluation metrics for unsupervised models with metrics like cluster distance.

5.5.8 LLMs

Better support for LLMs. The current solution do support the usage of LLMs. However, more can be done. This would most likely not mean setting up serving of models, but rather focusing on functionality to improve the LLMs context and memory, or potentially monitor sub-tasks when using techniques such as step-by-step thinking.

Furthermore, setting up a feedback system that can recorded if a LLM response was liked or disliked could add valuable insights, that would be useful for the ML system as a whole. Similar to what is possible with active learning. Generating a small test set for difficult examples, or examples where the output was not as expected could benefit the ML system as a whole and increase the quality. Lastly, evaluating LLMs can be tricky as they often do not follow a deterministic evaluation logic. Therefore, setting up evaluation metrics with multiple sub-requirements could be an interesting solution. E.i setting up a multi classification agent that evaluates the LLMs response based on sub criteria's.

Bibliography

Bach, S. H., Rodriguez, D., Liu, Y., Luo, C., Shao, H., Xia, C., Sen, S., Ratner, A., Hancock, B., Alborzi, H., Kuchhal, R., RÃ©, C., and Malkin, R. (2018). Snorkel drybell: A case study in deploying weak supervision at industrial scale.

Baeza-Yates, R., Ribeiro-Neto, B., de Araújo Neto Ribeiro, B., and ..., B.-Y.-R.-N. (1999). *Modern Information Retrieval*. ACM Press Books. ACM Press.

Bengio, Y., Louradour, J., Collobert, R., and Weston, J. (2009). Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, page 41â48, New York, NY, USA. Association for Computing Machinery.

Budelman, K. (2019). Three principles for designing ml-powered products.

Chaudhary, A. (2020). The illustrated self-supervised learning. `https://amitness.com/2020/02/illustrated-self-supervised-learning`.

Databricks (2022). What is mlops?

Drucker, H. and Cortes, C. (1995). Boosting decision trees. In Touretzky, D., Mozer, M., and Hasselmo, M., editors, *Advances in Neural Information Processing Systems*, volume 8. MIT Press.

Dutta, P., Cheuk, M. K., Kim, J. S., and Mascaro, M. (2019). Automl for contextual bandits. *CoRR*, abs/1909.03212.

Gardner, M. W. and Dorling, S. (1998). Artificial neural networks (the multilayer perceptron)âa review of applications in the atmospheric sciences. *Atmospheric environment*, 32(14-15):2627–2636.

Hosmer Jr, D. W., Lemeshow, S., and Sturdivant, R. X. (2013). *Applied logistic regression*, volume 398. John Wiley & Sons.

Huyen, C. (2022). *Designing Machine Learning Systems: An Iterative Process for Production-Ready Applications*. O'Reilly Media, Incorporated.

Klaise, J., Van Looveren, A., Cox, C., Vacanti, G., and Coca, A. (2020). Monitoring and explainability of models in production.

Kleppmann, M. (2017). *Designing Data-intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly Media.

Kreuzberger, D., Kühl, N., and Hirschl, S. (2022). Machine learning operations (mlops): Overview, definition, and architecture. *arXiv preprint arXiv:2205.02302*.

Li, Y. (2017). Deep reinforcement learning: An overview. *arXiv preprint arXiv:1701.07274*.

Likas, A., Vlassis, N., and J. Verbeek, J. (2003). The global k-means clustering algorithm. *Pattern Recognition*, 36(2):451–461. Biometrics.

Maas, A., Daly, R. E., Pham, P. T., Huang, D., Ng, A. Y., and Potts, C. (2011). Learning word vectors for sentiment analysis. In *Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies*, pages 142–150.

Meng, Q., Catchpoole, D., Skillicom, D., and Kennedy, P. J. (2017). Relational autoencoder for feature extraction. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 364–371. IEEE.

Mitchell, M., Wu, S., Zaldivar, A., Barnes, P., Vasserman, L., Hutchinson, B., Spitzer, E., Raji, I. D., and Gebru, T. (2019). Model cards for model reporting. In *Proceedings of the conference on fairness, accountability, and transparency*, pages 220–229.

Mollestad, M. E. (2023). Understanding the chaotic landscape of mlops.

Ng, A. (2021). A chat with andrew on mlops: From model-centric to data-centric ai. `https://www.youtube.com/watch?v=06-AZXmwHjo`.

Northcutt, C. G., Jiang, L., and Chuang, I. L. (2022). Confident learning: Estimating uncertainty in dataset labels.

Rabanser, S., GÃ¼nnemann, S., and Lipton, Z. C. (2018). Failing loudly: An empirical study of methods for detecting dataset shift.

Rafferty, G. (2020). A/b testing â is there a better way? an exploration of multi-armed bandits.

Ribeiro, M. T., Wu, T., Guestrin, C., and Singh, S. (2020). Beyond accuracy: Behavioral testing of nlp models with checklist.

Schubert, E., Sander, J., Ester, M., Kriegel, H. P., and Xu, X. (2017). Dbscan revisited, revisited: why and how you should (still) use dbscan. *ACM Transactions on Database Systems (TODS)*, 42(3):1–21.

Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J.-F., and Dennison, D. (2015). Hidden technical debt in machine learning systems. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'15, page 2503â2511, Cambridge, MA, USA. MIT Press.

Settles, B. (2009). Active learning literature survey.

Sommerville, I. (2004). *Software Engineering: Seventh Edition*. Pearson Education.

Suraj Patil, Pedro Cuenca, N. L. P. v. P. (2022). Stable diffusion with diffusers. `https://huggingface.co/blog/stable_diffusion`.

Tenenholtz, M. (2022). Hereâs the fatal error they made that you must avoid when deploying models.

Times, T. N. Y. (2022). Zillow, facing big losses, quits flipping houses and will lay off a quarter of its staff.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need.

William T. Freeman, M. R. (1994). Orientation histograms for hand gesture recognition. Technical Report TR94-03, MERL - Mitsubishi Electric Research Laboratories, Cambridge, MA 02139.

Appendices

5.6 Interviewees for Background

The following people was interviewed regarding the current state of ML, and the pain points related to deploying ML systems.

Table 5.1: Interviewees regarding state of ML

Date of Interview	Role
24th of November	Phd in Applied ML
29th of November	Senior ML Engineer
29th of November	Big Data Engineer
6th of December	Architecture Lead

5.7 Background theory interview template

Intro to who they are

Which type of AI do you have experience with? E.g: Edge computing, Realtime, Batch computing, Deep Learning, etc.

Any challenges that you find to be unique when deploying ML, compared to "regular" software deployment?

How do you keep the models consistent with between serving and training?

What do you find to be the biggest challenge when deploying ML?

How do the versioning work?

According to Hidden technical dept in ML Systems, is there a lack of good ML abstractions, so what do you find to be the best for ML?

How is the data handled in the model

Do culture effect the deployment of ML models?

How do business strategy effect ML models or vis versa?

What do you see as the biggest reason for ML projects that fail?

Is there something ML systems can learn from BI systems?

5.8 Interviewees for the proposed solution

The following people was interviewed regarding the proposed solution described in section 4.2.

Table 5.2: Interviewees regarding the proposed solution

Date of Interview	**Role**
6th of May	Lead Data Scientist
20th of May	Master student in AI
20th of May	Master student in AI
23th of May	ML Engineer
1th of June	Master student in AI
10th of June	Phd in Applied ML