Master's thesis

Master's Programme in Computer Science

Comparison of Two Open Source Feature Stores for Explainable Machine Learning

Tintti Rahikainen

February 12, 2023

Faculty of Science

University of Helsinki

Contact information

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki,Finland

Email address: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Study programme
Faculty of Science	Master's Programme in Computer Science

Tekijä — Författare — Author

Tintti Rahikainen

Työn nimi — Arbetets titel — Title

Comparison of Two Open Source Feature Stores for Explainable Machine Learning

Ohjaajat — Handledare — Supervisors

D.Sc. Mikko Raatikainen, D.Sc. Saku Suuriniemi

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's thesis	February 12, 2023	34 pages

Tiivistelmä — Referat — Abstract

Machine learning operations (MLOps) tools and practices help us continuously develop and deploy machine learning models as part of larger software systems. Explainable machine learning can support MLOps, and vice versa. The results of machine learning models are dependent on the data and features the models use, so understanding the features is important when we want to explain the decisions of the model.

In this thesis, we aim to understand how feature stores can be used to help understand the features used by machine learning models. We compared two existing open source feature stores, Feast and Hopsworks, from an explainability point of view to explore how they can be used for explainable machine learning.

We were able to use both Feast and Hopsworks to aid us in understanding the features we extracted from two different datasets. The feature stores have significant differences, Hopsworks being a part of a larger MLOps platform, and having more extensive functionalities.

Feature stores provide useful tools for discovering and understanding the features for machine learning models. Hopsworks can help us understand the whole lineage of the data – where it comes from and how it has been transformed – while Feast focuses on serving the features consistently to models and needs complementing services to be as useful from an explainability point of view.

ACM Computing Classification System (CCS)

Software and its engineering → Software notations and tools → Software maintenance tools

Software and its engineering → Software notations and tools → Software libraries and repositories

Avainsanat — Nyckelord — Keywords

MLOps, feature stores

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Software study track

Contents

1 Introduction **1**

2 Background **4**

2.1 Explainable AI (XAI) 4

2.2 Feature store 7

3 Research method **11**

4 Overview of the selected feature stores **14**

4.1 Feast 14

4.2 Hopsworks 16

4.3 Summary 17

5 Results **19**

5.1 Feast 19

5.2 Hopsworks 21

5.3 Comparison 25

6 Discussion **29**

7 Conclusions **31**

Bibliography **32**

1 Introduction

Machine learning systems are systems that include machine learning models as a part of larger software systems. Those software systems often rely on DevOps practices for continuous delivery. As with developing traditional software and using DevOps (ISO, 2022) to continuously deliver the software, machine learning operations (MLOps) has been developed to answer the specific needs of machine learning systems in addition to the practices used in DevOps (John et al., 2021). Machine learning systems accumulate technical debt much like traditional software systems, but with additional machine learning specific problems that exist at the system level rather than code. For example, when the data inputted into a machine learning model has underutilised dependencies – input signals that are of little or no importance to the model. They can make the model more vulnerable to change and could be removed with no damage to the model performance (Sculley et al., 2015). This system-level technical debt can be hard to detect and lead to machine learning systems being expensive and difficult to maintain (Sculley et al., 2015). MLOps tools and practices can help detect and pay off this machine learning specific technical debt.

Organisations can be divided into three categories based on their machine learning maturity: *data-centric*, *model-centric*, and *pipeline-centric* (Mäkinen et al., 2021). Data-centric organisations are focused on how to manage and utilise data, and model-centric organisations on building and deploying their first model. Pipeline-centric organisations already have business-critical models in production, and they are focused on scaling and continuous development and maintaining the quality of the existing models. With organisations having increasing data and machine learning model maturity, continuously improving and redeploying models becomes relevant. MLOps is the automation of the steps needed to develop and deploy machine learning models, and with the continuous delivery of machine learning systems, it becomes more and more important.

Explainable artificial intelligence (XAI) is a field that has been growing recently due to the transparency and trust concerns about artificial intelligence (Adadi and Berrada, 2018). It is an interdisciplinary field striving to create methods to explain the decisions of systems using artificial intelligence, and the data driving those decisions. Many of the recent advances in artificial intelligence are in the field of machine learning, and machine learning systems have become increasingly common in our society, but especially deep

learning algorithms can be opaque in their functioning. For example, the recent language model GPT-3, which uses deep learning, uses 175 billion learning parameters (Floridi and Chiriatti, 2020) – far more than a human can reasonably keep track of. Explainability increases trust in machine learning systems by making them easier to control and improve, and making the decisions the system makes justifiable. XAI also aims not to lose the accuracy of the sophisticated machine learning systems while enabling trust and control of them.

While explainability and increasing trust in a machine learning system can be a goal in itself, it can also be a tool that helps us with continuous improvement and deployment of models. Explainability can give us insight into how to maintain and improve our models. Monitoring the data and decisions the model makes helps us discover bugs and broken integrations.

A central concept in machine learning systems is *features*, which are the input data for a machine learning model. Raw data is rarely inputted directly into the model, but features for the model are extracted from the data by transforming it, for example by scaling the values or combining different variables. The data used by machine learning models – both for training and inference – is an important aspect of explainable machine learning. The explanations of the machine learning systems' reasoning depend on the features the system relies on.

Feature stores can help us trace the features back to original data even after transformations – making the explanations easier to understand – and give us more insight into the features and data the model uses. A feature store is a system for transforming, storing and serving feature data for machine learning models. Feature stores were developed to manage and standardise the workflow in an end-to-end machine learning pipeline (Orr et al., 2021). A machine learning pipeline consists of ingesting the training data, training and deploying the models, and maintaining and monitoring them. Feature stores also store *metadata* of the features. Metadata can be any additional data about the data that is not included in the dataset, such as the creator of the dataset, descriptions of features, or the units that are used for different values.

In this thesis project, we studied how existing feature stores can help with explainable machine learning – how feature stores can be used to explain the features used by machine learning models and the data behind them. We chose two open source feature stores, Feast and Hopsworks, and compared using them in practice to explain different aspects of the features. We formulated the following research question:

How can feature stores help explain the features used for machine learning models?

We found that feature stores can be used to help understand feature data in many ways. Definitions and other metadata for features can be used to relay information to multiple teams working on the same data. Features can be traced back to the raw data they were extracted from and their use in different machine learning models can be tracked. Feature stores can be used to validate and monitor the feature data.

Concerning the tools, Feast is a more minimal tool for creating a separate data layer for the features, while Hopsworks provides more functionalities for understanding the features and data behind them. Hopsworks feature store can be used to serve different versions of the features, and run transformation functions on the features while automatically keeping track of the transformations. Hopsworks automatically computes statistics on the features, and an external library for validating features can be integrated into both Feast and Hopsworks.

This thesis is organised as follows: Chapter 2 introduces the concepts of explainability and feature stores in general. Chapter 3 goes into the methods used in this thesis. Chapter 4 introduces the tools used in the comparison, Feast and Hopsworks. The results of testing and comparing the feature stores are presented in Chapter 5. Chapter 6 discusses the results, and the conclusions are summarised in Chapter 7.

2 Background

2.1 Explainable AI (XAI)

Many recent advances in artificial intelligence are due to machine learning and especially deep learning models. These models are often very opaque and might make even important decisions in our lives, therefore the field of explainable artificial intelligence (XAI) has gained popularity (Adadi and Berrada, 2018).

There is no clear, all-encompassing definition for explainable AI. XAI refers to a range of techniques with the goal of making artificial intelligence more transparent. There are numerous terms related to and even used synonymously with explainability – such as interpretability, understandability and comprehensibility (Adadi and Berrada, 2018). XAI is used in various different domains, like transportation, military, healthcare, law, and finance – anywhere where AI makes decisions or predictions of profound human impact and it is important to know the grounds for them.

Adadi and Berrada (Adadi and Berrada, 2018) identify two organisations as the most prominent in XAI research: Fairness, Accountability, and Transparency in Machine Learning (FAT), and the Defense Advanced Research Projects Agency (DARPA). FAT has developed the "Principles for Accountable Algorithms" that define explainability as ensuring that "algorithmic decisions as well as any data driving those decisions can be explained to end-users and other stakeholders in non-technical terms" (Diakopoulos et al., 2016). DARPA's XAI program was launched in May 2017, and was organised into three research areas: developing new XAI machine learning and explanation techniques, understanding the psychology of explanation, and evaluating the new XAI techniques (Gunning et al., 2021). They define explainable AI as "AI systems that can explain their rationale to a human user, characterize their strengths and weaknesses, and convey an understanding of how they will behave in the future." (Gunning and Aha, 2019) In their XAI program DARPA also found that even though there was a tension between learning performance and explainability, explainability can also improve performance, as producing explanations encouraged systems to learn more effective representations of the world (Gunning et al., 2021).

Based on the survey papers by Adadi and Berrada (Adadi and Berrada, 2018), and Guidotti et al. (Guidotti et al., 2018) there are multiple motivations for explainable in AI systems, and multiple ways to categorise and characterise different explainability methods.

The need for explainable AI can be divided into four reasons: *justification*, *control*, *improvement*, and *discovery* (Adadi and Berrada, 2018). When explaining to justify the goal is to explain the reasoning or justification behind a certain decision an AI system has made, rather than the decision-making process of the system in general. Other than for justifying the decisions of AI models, explainability can also be used for finding vulnerabilities and debugging (explaining for control). Explainability can also make continuously improving models easier. Understanding how a model comes to a specific conclusion might also give us new insight into the problem we are solving, or the data we are using. Explaining models to justify their decisions, enabling control over them and continuously improving them increases the trust in the systems using these models.

Adadi and Berrada classify different methods used to explain machine learning models by three criteria: *complexity*, *scope*, and *dependency on the model*. These are described in the following paragraphs.

The classic approach to explainability is making the machine learning models themselves explainable on their own, that is, making them less complex. These kinds of models include for example linear models, decision trees, and rules-based systems. A common challenge in this approach is the tradeoff between model accuracy and interpretability. For example, the listed models are only explainable when they are reasonably sized for a human to understand (Guidotti et al., 2018). Another approach is to make a complex black-box machine learning model and create a method for coming up with the explanation afterwards. There are also methods that modify the internal structure of a complex machine learning method to make it more interpretable (Adadi and Berrada, 2018).

The scope of the explanation can either be *local* or *global*. Local methods explain the reasons for specific decisions or predictions made by the machine learning model, and global methods aim to explain the underlying logic of the entire model, explaining the reasoning behind every possible outcome.

Model specific methods are dependent on the machine learning model and are only used to explain the model they were developed for. *Model agnostic* methods on the other hand can explain the reasoning of different kinds of machine learning models. An example of a model agnostic, local explanation method for a black-box machine learning model is LIME (Ribeiro et al., 2016). It learns an approximate interpretable model locally around

a prediction and discovers which features were important for that prediction.

Guidotti et al. identify three dimensions of interpretability: *scope*, *time limitation*, and the *nature of user experience* (Guidotti et al., 2018). The scope can be global or local, as explained above. Time limitation means how much time the user has to understand the explanation – do they need a simple explanation that is easy to grasp or a more detailed explanation that requires more time to understand. The nature of user experience is also important when considering what kind of explanation a user needs. A data scientist debugging their model needs a different kind of transparency to the model's reasoning than for example a domain expert using the model to make decisions.

Data is crucial for the predictions of machine learning models, as datasets fundamentally influence the models' behaviour. Even when a model works as intended, using bad data to train a model can recreate and even amplify the problems in the data. An example of biased data amplifying societal problems is predictive policing – using statistical predictions to prevent and solve crime (Perry et al., 2013). Police records used to train machine learning models for predictive policing are biased, since areas where there is more police presence are over-represented in the data, and the predictive models end up increasing the number of police officers in already over-policed areas (Lum and Isaac, 2016).

Explainability for data used in machine learning is important so that we can find and recognise these biases. Many of the methods for data explainability centre around the documentation of datasets. *Datasheets for datasets* contain questions for the creator of the dataset grouped into motivation, composition, collection process, data processing, uses, distribution, and maintenance (Gebru et al., 2021). The workflow for creating the datasheet is intended to be manual and to encourage the creator of the dataset to reflect on the data while creating, distributing and maintaining the dataset. *Data statements* are a documentation practice developed for data used in natural language processing (Bender and Friedman, 2018). *Data cards* were developed to complement long-form and domain-specific documentation practices like datasheets and data statements (Pushkarna et al., 2022).

In summary, XAI is a set of techniques and practices that aim to make AI systems more transparent. Modern machine learning systems can have billions of learning parameters and be very non-transparent. Explainability makes machine learning systems easier to control and improve, and decisions based on their predictions justifiable. Different explainability methods can be based on making the models less complex or creating a method for making explanations for black-box models. They can be functional with only a certain

type of model, or be model agnostic and able to explain any model's predictions. When making explanations, there are different dimensions to consider – the scope of the explanation, the time that the user has for interpreting the explanation, and the nature of the user's experience. Data is a crucial part of machine learning systems, and explaining the data used for the machine learning models is vital for both understanding the functioning of the model and recognising problems in the data itself.

2.2 Feature store

Feature stores are tools used for transforming, storing, serving, and monitoring features for machine learning models (Orr et al., 2021). They run transformations on raw data ingested from batch or streaming sources to create features to serve models. The features are stored in the feature store, which serves as a central repository of features that can be reused across teams. Features can be published in the feature store with a description and other metadata that helps find and use them in different teams across an organisation. Feature stores also consistently serve features for model training and inference, and support monitoring the data for, e.g., data skew.

The functions of a feature store are depicted in Figure 2.1. Data is ingested from either a streaming or batch source, and transformed, stored and monitored in the feature store. Data scientists can define features, and search and discover existing features from the feature store. They can fetch training datasets from the feature store to train their models with. For deployed models making real-time predictions, the feature store works as a feature server from which the feature vectors used for predictions can be fetched.

The first industrial feature store was developed at Uber in 2017 (Hermann and Del Balso, 2017) as a part of Uber's machine learning platform Michelangelo. Since then there has been a number of different feature stores developed.

The industrial *machine learning pipeline* has three steps (Orr et al., 2021):

1. Cleaning, checking, and extracting features from training data acquired from possibly multiple sources.
2. Training and deploying models using the features extracted from data
3. Monitoring and maintaining the deployed models

Each of the steps has challenges that feature stores can help with.

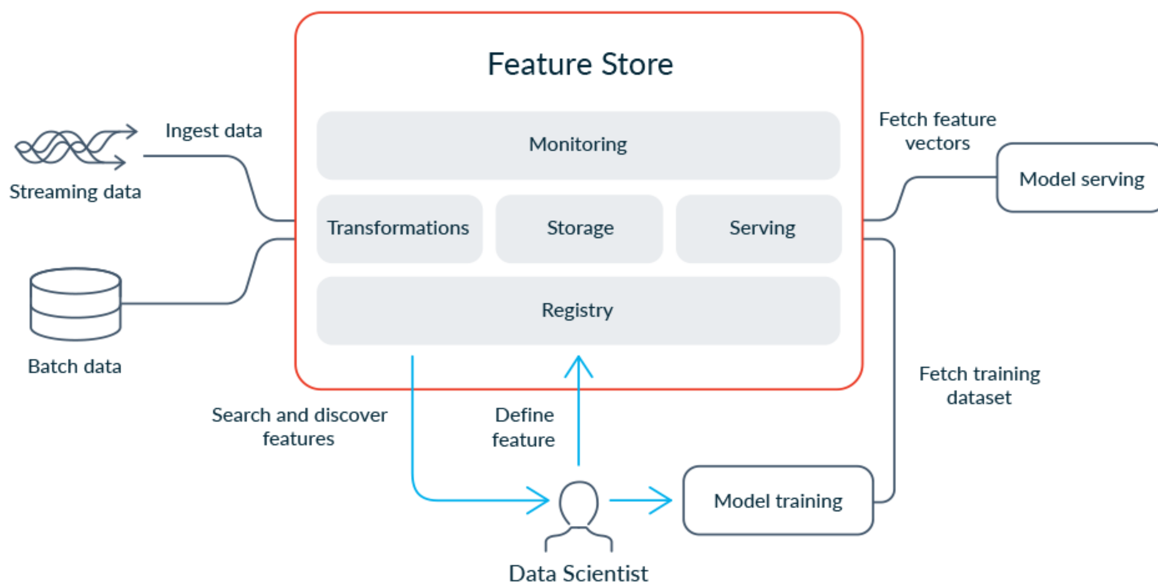


Figure 2.1: A feature store can be used to transform, store, serve, and monitor feature data. It enables the search and discovery of features across teams, and consistently serves features for model training and inference. (Del Balso, 2020).

Concerning training data, data is often, if not always, transformed in some ways before inputting it into the machine learning model to improve the model performance – this is called feature engineering. A common transformation for example is *standard scaling*, where the features are transformed to have zero mean and unit variance. Especially in a large organisation with data scientists working on multiple problems using the same data can lead to duplicated work when they need to do similar feature engineering tasks on the data. This can also lead to a lack of consistency in feature definitions (Orr et al., 2021). Feature stores provide a centralised repository of reusable features that are accessible to all data scientists, which reduces duplicate work. Raw data can be messy and especially if the features are combined from multiple sources it can be difficult to tell what certain columns actually mean. Feature stores include a data catalogue, or a central registry, where all data scientists can find features with standardised definitions and metadata (Del Balso, 2020).

Concerning training and deploying models, feature stores often consist of online and offline data storage for storing data for model training and inference. The offline storage is for training the model, and the online storage is for serving features for the deployed model. The deployed machine learning models need to be served features for their predic-

tions, and for the real-time serving of features the same transformations need to be done to new data as were done to the training data. Differences in the transformations can cause *training-serving skew* – a difference in the performance of the model in training and production (Bogner et al., 2021).

Concerning monitoring and maintaining models, feature stores can calculate metrics on the features they serve, and help detect issues in the data. Feature stores can also be used to track and monitor operational metrics by logging for example feature availability, capacity, staleness, throughput, latency, error rates, etc. (Del Balso, 2020) The data might come from multiple different integrated systems, and the more integrations there are the more probable is that some of the integrations break. Monitoring the ingested data and features can help discover these broken integrations. If the data or features change, for example when a bug in data ingestion or transformation is found and fixed, it might be challenging to communicate the changes to all the data scientists using the data. As a feature store provides a centralised data catalogue as a single source of truth for the features, it makes communicating these changes easier. Training-serving skew can be easily monitored by comparing the features in the feature store's online storage to the training features in the offline storage.

Often there might be a need to analyse historical data. In feature stores, data can be timestamped and versioned, so it is possible to analyse the model's behaviour as it was, e.g., 30 days ago. Versioning allows the user to create and store different versions of the data and retrieve the different versions as needed. Versioning of code, data and environments, and provenance information makes reproducing the execution of specific machine learning pipelines possible – and also makes it possible to tell whether the current setup will provide similar results to the original (Ormenisan et al., 2020).

Data lineage connects the features to the original data and tells us how the data has been transformed to result in the features we have. If we are trying to explain the reasoning or results of the machine learning model, the features themselves might not give good insight – for example, a standardised value can be nothing like the original value and can seem like nonsense to a human.

Existing tools make feature stores more accessible to organisations. It takes an organisation with quite advanced data maturity to recognise or even have the need for feature stores or other more advanced data infrastructure. Existing tools make it easier and less expensive to adopt new practices and infrastructure for data.

In summary, the essential functionalities of a feature store are the transformation, storage,

serving and monitoring of feature data for machine learning models. Consistent transformations for training and serving data help prevent performance differences in training models and inference in production, and keeping track of the transformations helps understand the features. Feature stores work as a registry for finding different features. Feature definitions and other metadata help understand the features' meanings and give visibility to why the features were created and by whom. Feature stores also store the information on which data sources the features are from, and help trace the features to correct data sources when multiple data sources are combined to create sets of features. Versioning the feature data gives visibility to models using modified versions of the features. Feature stores enable the monitoring of stored features to discover problems such as broken integrations or training-serving skew.

3 Research method

In this thesis we tested two feature stores to explore how they can be used to help explain the features used for machine learning models, as explaining the features can help with any of the objectives of explainability: controlling, improving and justification of machine learning models and their predictions. Based on Chapter 2 we chose seven functionalities of feature stores to explore how they can help with explainability:

1. **Feature definitions.** How can we have more visibility into what the features stored actually mean? (→ Storage)
2. **Standardised metadata.** When multiple teams are working together on the same data, how can we ensure the features have all the necessary metadata available with them? (→ Storage)
3. **Tracking transformations.** How can we find the original values of transformed features? (→ Transformation)
4. **Combining multiple data sources.** How can we tell which data source is a feature from when using multiple data sources for features? (→ Serving)
5. **Tracking feature use.** In the case of, e.g., sensitive data, how can we track which models are using certain data? (→ Serving)
6. **Feature versioning.** When features are used across teams, how can we ensure the correct version of the features is used? (→ Serving)
7. **Data validation and monitoring.** How can we validate and monitor the data, so possible problems with it are noticed? (→ Monitoring)

The feature stores compared in this thesis are Feast (Feast, 2022) and Hopsworks (Logical Clocks, 2022). We chose these feature stores because they are open source tools that are readily available for anyone to use. Both support a number of different cloud platforms for storing data, as well as on-premise storage. They also represent different approaches to feature stores. Feast is a very lightweight stand-alone feature store, and the Hopsworks

feature store is part of a larger MLOps platform. Feast and Hopsworks are introduced in more detail in Chapter 4.

We carried out analysis as follows. We read the documentation of both tools to familiarise ourselves with the tools, and to get an understanding of what the tools can be used for and how. We installed Feast version 0.26.0 and Hopsworks version 3.0 on a laptop running macOS 12.6 and tested them in practice.

We used two different datasets to test the feature stores: a dataset of spectral data on stars (Ku, 2020), hereafter "Star dataset", and a time series dataset on environmental sensor telemetry data (Stafford, 2020), hereafter "Sensor dataset". For the purposes of this thesis, the data was stored locally as csv and parquet files. We divided the Sensor dataset into two separate datasets to test combining datasets in the feature stores. Each dataset had different sensor readings, as well as the timestamp of the reading and the sensor name to use for combining the data later. Figure 3.1 shows a sample of the split Sensor dataset.

```
             ts             device        co  light     smoke       lpg
0  1.594512e+09  b8:27:eb:bf:9d:51  0.004956  False  0.020411  0.007651
1  1.594512e+09  00:0f:00:70:91:0a  0.002840  False  0.013275  0.005114
2  1.594512e+09  b8:27:eb:bf:9d:51  0.004976  False  0.020475  0.007673
3  1.594512e+09  1c:bf:ce:15:ec:4d  0.004403   True  0.018628  0.007023
4  1.594512e+09  b8:27:eb:bf:9d:51  0.004967  False  0.020448  0.007664

             ts             device   humidity  motion       temp
0  1.594512e+09  b8:27:eb:bf:9d:51  51.000000   False  22.700000
1  1.594512e+09  00:0f:00:70:91:0a  76.000000   False  19.700001
2  1.594512e+09  b8:27:eb:bf:9d:51  50.900000   False  22.600000
3  1.594512e+09  1c:bf:ce:15:ec:4d  76.800003   False  27.000000
4  1.594512e+09  b8:27:eb:bf:9d:51  50.900000   False  22.600000
```

Figure 3.1: Sample of the Sensor dataset split into two dataframes.

Testing Feast. Feast requires the features ingested to have timestamps, so we generated dummy timestamps for each row in the Star dataset. When using a file as offline storage instead of a data warehouse or data lake, Feast only supports parquet files, so the original csv file was also converted to a parquet file.

We defined the features and feature views for the Star dataset and different feature services for testing feature retrieval. We retrieved the feature data for training sets using the feature services and saved them as datasets. We tested the data validation for historical features with the data validation tool Great Expectations (Great Expectations, 2023) with simple validation rules.

The Sensor dataset already had timestamps so generating them was not necessary, but

we converted the original csv files to parquet. We defined feature views for both parts of the Sensor dataset, and an entity based on the sensor name to be used for combining the separate feature views into a single feature service.

Testing Hopsworks. We used the Hopsworks serverless platform for testing the feature store, so we did not have to set up the infrastructure for the data. Hopsworks requires feature groups to have a primary key, so those were added for the Star dataset since it had no unique identifier for each star. We generated a sequence of integers for the primary keys.

We created the feature group for the Star dataset and ingested the data from the csv file to Hopsworks with the Hopsworks feature store Python API. We added a standard scaler as a transformation for some of the features in the Star dataset and created training datasets with the feature views and transformations in Hopsworks.

The Sensor dataset also needed a unique primary key for each of the samples, so we generated a sequence of integers for the primary keys applied before the dataset was split into two and ingested into Hopsworks. We ingested the Sensor dataset into two separate feature groups and combined them into a single feature view.

4 Overview of the selected feature stores

For the comparison for this thesis, we chose two open source feature stores: Feast (Feast, 2022) and Hopsworks (Logical Clocks, 2022). The version and licence of the tools tested, and platforms supported for hosting the tools and data are presented in Table 4.1.

Table 4.1: Feature stores included in the comparison.

Feature store	Feast	Hopsworks
Version	0.26.0	3.0
Licence	Apache License 2.0	AGPL-3.0
Supported platforms	On-Prem, AWS, GCP, Azure (alpha)	On-Prem, AWS, GCP, Azure
URL	https://feast.dev/	https://www.hopsworks.ai/

4.1 Feast

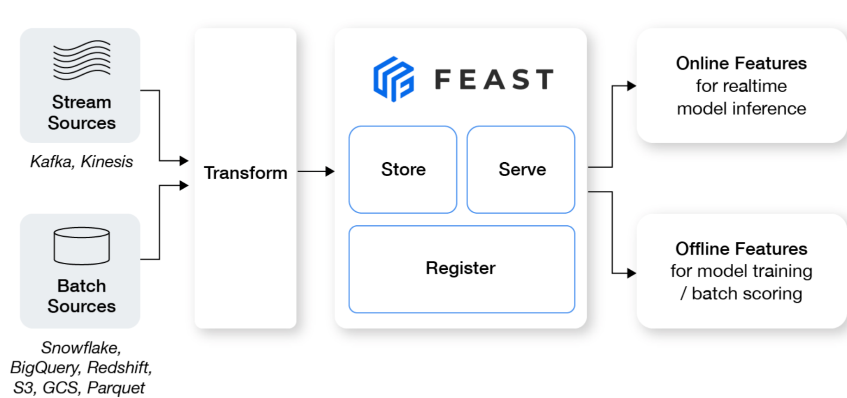
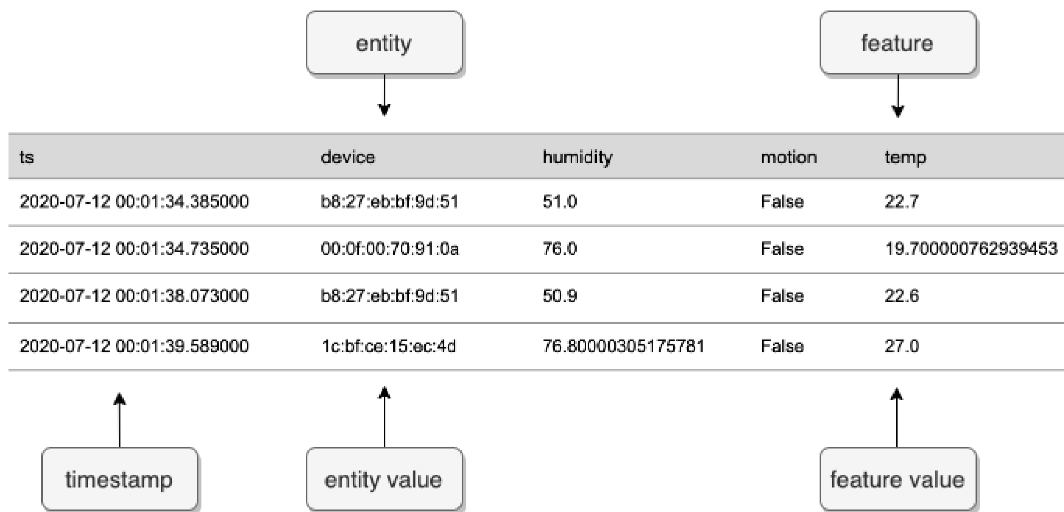


Figure 4.1: Features are ingested into Feast from stream or batch sources. Features are managed in Feast, and served as online features for real-time inference as well as offline features for model training and batch scoring. (Feast, 2022).

Feast is a minimal open source feature store. It was originally developed by Gojek and Google Cloud to help maintain and serve features across multiple teams. Feast was open sourced in 2018, and is now governed by The Linux Foundation (Sell and Pienaar, 2019).

A Feast *project* defines a namespace that is isolated from other projects so that the user can not retrieve features from multiple projects with a single call. It is recommended that a single project and feature store is used for each environment – such as the development, staging, and production environment (Feast, 2022). In a project, the user can define *feature views* that contain the features. For example, the features in the Star dataset form a single feature view. The features can relate to *entities*, for example, a single sensor that has given multiple readings over a period of time, as seen in Figure 4.2. Entities are used for retrieving features, by providing an entity key to fetch the features related to that entity. Features have *data sources* that can be for example a parquet file or a table in a database. For retrieving the features, *feature services* can be used to represent a group of features from one or more feature views.



ts	device	humidity	motion	temp
2020-07-12 00:01:34.385000	b8:27:eb:bf:9d:51	51.0	False	22.7
2020-07-12 00:01:34.735000	00:0f:00:70:91:0a	76.0	False	19.700000762939453
2020-07-12 00:01:38.073000	b8:27:eb:bf:9d:51	50.9	False	22.6
2020-07-12 00:01:39.589000	1c:bf:ce:15:ec:4d	76.80000305175781	False	27.0

Figure 4.2: Features in Feast can relate to entities, such as an individual sensor in the sensor dataset. (Adapted from Feast, 2022).

Feast is developed primarily to support timestamped tabular data, and it expects all data to be timestamped. For data that does not have timestamps, they need to be generated. Data can be ingested from *batch* or *stream sources*. Batch sources include data warehouses and data lakes. Feast does not have native integrations for streaming sources, but streaming data can be ingested either by pushing data into Feast or directly from Kafka or Kinesis, although direct ingestion is still an experimental functionality.

The flow of the feature data in Feast is depicted in Figure 4.1. Feature data is ingested from stream or batch sources and transformed. Feast manages the features and stores the feature definitions. Features are served using the feature services, either as offline features for model training and batch scoring – making a larger batch of inference on the data –

or online features for real-time inference.

Feast configuration is written declaratively as code in a *feature repository*. The feature repository consists of Python files containing the feature declarations, an infrastructural configuration file for the project, and a .feastignore file if the feature repository needs to ignore certain paths. The Feast CLI then uses the feature repository to configure, deploy, and manage the feature store (Feast, 2022). Feast also has a web UI that allows users to browse data sources, feature views, entities and feature services. The web UI is still an experimental functionality of Feast.

4.2 Hopsworks

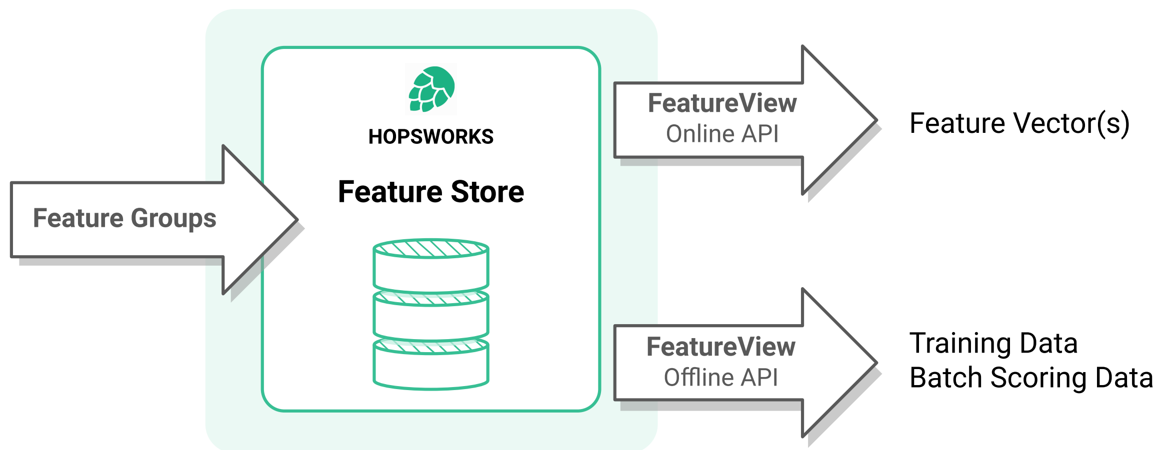


Figure 4.3: Features are ingested into Hopsworks as feature groups, stored in the Hopsworks feature store, and served as feature views via either the offline or online API. (Logical Clocks, 2022).

The Hopsworks Feature Store was the first open source feature store developed, and it is part of a larger MLOps platform. The feature store can be also used separately from the more general MLOps platform. Hopsworks was first developed at KTH Royal Institute of Technology as a part of a research platform and later by Logical Clocks, which was founded by a group of researchers at KTH (Dowling, 2018).

The features in Hopsworks Feature Store are organised into *feature groups* that are tables of features with a primary key and optionally a timestamp and partition key. The partition key can be used to efficiently query features from the feature store and accessed via an API (Logical Clocks, 2022).

The feature groups are stored either in an online or offline store. The online store only

stores the latest values of the feature group, while the offline store stores the historical data. Typically the online data is used for real-time serving for models, and offline data for creating training data for models and making predictions in batches (Logical Clocks, 2022). Feature groups can be stored in Hopsworks or externally as external feature groups that are accessed through a connector API.

Retrieving features in Hopsworks is done through *feature views*. A feature view consists of features from one or more feature groups, and user-defined transformation functions written in Python that are applied to individual features. Multiple feature groups can be combined into a single feature view by querying the feature groups and joining the queries on any join key specified. Hopsworks queries support inner, outer, left, and right joins.

Figure 4.3 presents an overview of the flow of the feature data in Hopsworks. Features are ingested into feature groups and stored in the Hopsworks feature store. They are served using the feature views as either offline features for model training and batch scoring or online feature vectors for real-time inference.

4.3 Summary

Table 4.2 summarises the main differences between Feast and Hopsworks. Feast is a minimal, standalone feature store, and Hopsworks is a larger MLOps platform that includes a feature store that can also be used separately. With Feast, all the feature data is stored externally, and Feast provides a separate data layer for managing and serving feature data. Hopsworks provides the infrastructure to store the feature data, although features can also be stored externally.

Feast and Hopsworks have some differing requirements for the data – Feast requires the data to be timestamped, whereas Hopsworks requires a unique primary key for each row of the data. Combining the data is also approached differently in both of the feature stores. In Feast, the features can relate to an entity that is then used to combine the features in different feature views. In Hopsworks the feature data is fetched with queries that can be combined with joins using any existing key.

Table 4.2: A summary of the differences between Feast and Hopsworks.

Feast	**Hopsworks**
Minimal standalone feature store	MLOps platform with a feature store that can also be used separately
Feature data stored externally	Feature data stored internally or externally
Requires feature data to be timestamped	Requires feature data to have unique primary keys
Features organised into *feature views*	Features organised into *feature groups*
Feature services used for combining multiple data sources and fetching features	*Feature views* used for combining multiple data sources and fetching features
Entities used for combining multiple feature views into a single feature service	*Joined queries* used for combining multiple feature groups into a single feature view

5 Results

In this chapter, we present the results of testing Feast and Hopsworks, as described in Chapter 3.

5.1 Feast

Feature definitions. Features are defined in Feast with a feature name, data type, and optionally tags. The tags can be any key-value pairs of strings, so a description of the feature can be included in them.

Example. Features extracted from the Star dataset includes the absolute magnitude of a star. The feature is named `Amag` and a tag named `description` is added to it that contains the feature description in the feature view, as shown in Figure 5.1.

Standardised metadata. Any metadata can be included in the tags, but their use is not required in Feast, so in order to have standardised metadata across multiple teams creating the features, they need to be agreed on or enforced some other way. Figure 5.1 shows the properties and tags of a feature shown in the Feast Web UI. The web UI is still an experimental functionality of Feast.

Example. In the Sensor dataset, the metadata of the features includes the unit of the sensor reading. For example, the temperature is recorded in Fahrenheit, and a custom tag with the name `unit` can be added to the feature to indicate this.

Tracking transformations. Data transformation is done outside of Feast itself, so Feast does not keep track of the transformations done to data automatically. The exception to this is the still experimental on-demand feature views in Feast. On-demand feature views include transformations that are computed locally on online and offline data.

Example. In the Star dataset, the absolute magnitude of a star is calculated using the visual apparent magnitude and stellar parallax of the star, which are included in the dataset. Since the transformations happen outside of Feast, the feature has been created beforehand and the transformation needs to be tracked manually, for example in the feature tags.

Combining multiple data sources. Feature services can be combined from multiple

Feature: Amag

Overview

Properties

Name
Amag

Value Type
FLOAT

FeatureView
spectral_data

Tags

custom-tag
Custom tags can be any key-value pairs.

description
Absolute Magnitude of the Star. The absolute magnitude of the stars were generated via the equation: M = m + 5*(np.log10(p) + 1) Where M represents the absolute magnitude Amag, m represents the visual apparent magnitude Vmag and p represents stellar parallax Plx

Figure 5.1: Feature properties and tags displayed in the Feast Web UI.

feature views and data sources, and for each feature in the feature service it is shown from which feature view it originates. The feature views have information on what data source the features are from. Figure 5.2 shows the view for a feature service in Feast Web UI, including features from two different feature views.

Example. The split Sensor dataset was ingested into separate feature views `sensor_data1` and `sensor_data2`, and grouped into a singe feature service `sensor_data_service`. The Feast Web UI shows which features come from which feature views, as shown in Figure 5.2.

Tracking feature use. Feast's feature services allow for tracking where features are being used and by whom. They can also limit the features that can be retrieved from a dataset. If a feature in a feature view is not defined in the feature service used to retrieve data, it won't be returned from the feature store. This can help track which features are used in models, for example in a case where the data includes sensitive information.

Example. The feature service `sensor_data_service` for the sensor data pictured in Figure 5.2 has a tag named `owner` to indicate who created the feature service. Only the features listed in the feature service can be retrieved, if the dataset had other features they cannot be accessed through this feature service.

Feature versioning. Feast has no versioning. Separate versioning tools can be used with Feast, but their use is not covered in this thesis.

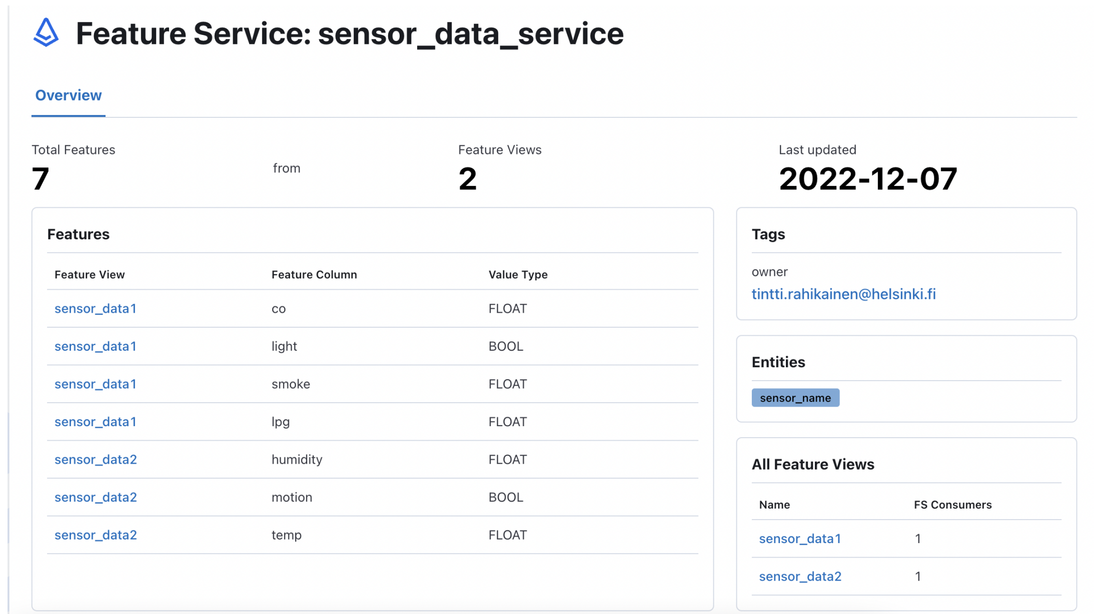


Figure 5.2: A feature service in Feast can be used to combine different data sources, track the usage of features, and control which features can be retrieved from the feature store. Features from two different feature views are combined in the pictured feature service.

Data validation and monitoring. Feast has experimental support for data validation and quality monitoring using Great Expectations (Great Expectations, 2023). The support is only for historical features, support for validating features when writing to and reading from online storage is planned. Validating offline features needs an existing saved dataset in Feast to use for reference for the Great Expectations expectation suite. Features are validated when retrieving historical features. If the validation fails, an exception is raised that contains details for the expectations that did not pass.

Example. In the Sensor dataset where we combined the split dataset into a single feature service, we used Great Expectations to validate that the feature service had all the required features from each part of the dataset. We created simple rules for the validation, and a saved training set to use as a reference for Great Expectations to validate feature data retrieved in the future.

5.2 Hopsworks

Feature definitions. Each feature has a name and a data type that can be declared in code or inferred from the ingested data. Features are grouped into feature groups, and

features and feature groups have optional descriptions.

Example. The Star dataset was ingested into Hopsworks and a name, data type, and description were added to each feature. Figure 5.3 shows some of the features in a feature group and their descriptions.

Standardised metadata. Feature groups can have custom tags and keywords, but individual features do not. Tags need to have a tag schema defined, which tells what keys and values are included in the tags.

Example. The Sensor dataset has metadata about the unit of each feature. Since the individual features do not have custom tags, the information about the unit was added to the feature description.

name	type	description
star_id	bigint	ID of the Star
vmag	double	Visual Apparent Magnitude of the Star
plx	double	Distance Between the Star and the Earth
e_plx	double	Standard Error of Plx (Drop the Row if you find the e_Plx is too high!)

Figure 5.3: Feature definitions in Hopsworks include the name, data type and description of the feature.

Tracking transformations. Transformations can be applied to the data inside the Hopsworks platform, and are tracked automatically. These transformations are only applied after fetching the features, so some heavier computations might be good to apply to the features beforehand to save time. Hopsworks has some built-in transformation functions, like the standard scaler, and custom transformation functions can also be written in Python and applied to features.

Example. The feature values in the Star dataset were scaled with a built-in standard scaler in Hopsworks. Figure 5.4 shows a list of features in a feature view, with the transformations applied to them listed next to the feature name. The code for the transformations can be

previewed and downloaded from Hopsworks, as shown in Figure 5.5. The transformations cannot be used to combine multiple features into one, so the absolute magnitude of the star is computed beforehand also in Hopsworks.

Figure 5.4: Transformations made in Hopsworks can be seen in the feature view list of features.

Figure 5.5: The transformation function used in Hopsworks can be previewed, copied, and downloaded. Hopsworks has some built-in transformation functions like the standard scaler pictured, and you can also add custom transformation functions written in Python.

Combining multiple data sources. Feature views are used to fetch data from Hopsworks. Feature views can consist of features from multiple feature groups, including external feature groups. The feature group of origin is listed with the feature in the feature view.

Example. The Sensor dataset split into `sensor_data_1` and `sensor_data_2` was combined into a single feature view `sensor_data_fv` in Hopsworks. The feature groups are listed along with the features in the feature view in the Hopsworks UI, as seen in Figure 5.6.

Tracking feature use. Feature groups automatically keep track of which feature views they are used in, as well as who created the feature view. In Hopsworks this information

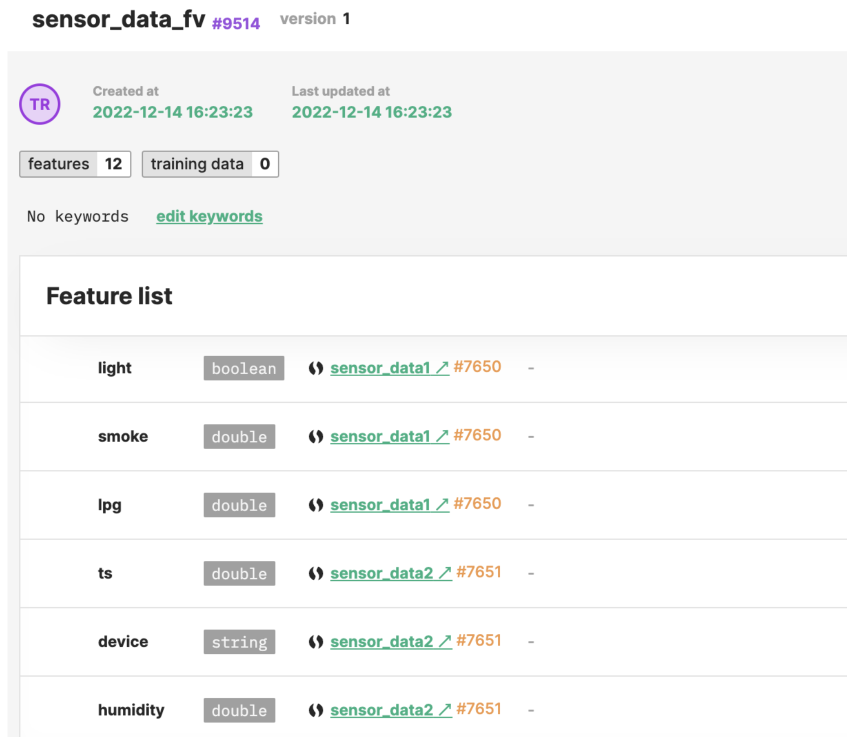


Figure 5.6: When combining multiple feature groups into a single feature view in Hopsworks, the feature group is displayed along with the feature.

is called *provenance* and can be found in the UI under each feature group.

Example. We created multiple feature views for the Star dataset, and they can be seen in Figure 5.7, which displays the provenance of the feature group `star_data`. Hopsworks also displays the owner of a selected feature view and information on when the feature view was last updated.

Feature versioning. Feature groups are versioned, and the user can specify which version of the feature group is used in the feature view they create. Feature views and the transformation functions for features are also versioned.

Example. We created additional versions of the feature groups from the Sensor dataset. When retrieving the data via a feature view, the version of the feature group used is defined so the correct version is used and the version used is not changed unexpectedly.

Data validation and monitoring. Hopsworks can automatically compute different statistics of the features stored in the feature store. Some basic statistics are enabled by default, and others can be enabled. By default, Hopsworks computes an approximate count of the distinctive values and the completeness, as well as the minimum, maximum,

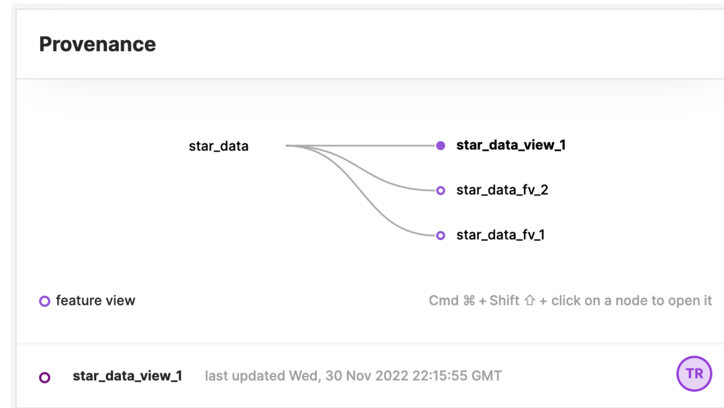


Figure 5.7: Hopsworks shows the data provenance of feature groups – which feature views use features from the feature group, and the creator of the feature view.

mean, standard deviation and the sum of each feature for numerical values. In addition, exact statistics that are more expensive to compute can also be enabled: exact count of distinctive values, entropy, uniqueness and distinctiveness of the value of a feature. Hopsworks can also display a histogram of the distribution of the feature values, as well as the correlation of the features.

Example. We enabled all the additional statistics for the star data in Hopsworks. Figure 5.8 shows the statistics and histogram of the absolute magnitude `amag` of the stars as displayed in the Hopsworks dashboard, and Figure 5.9 shows the correlation matrix. The correlation matrix also shows the precise correlation values of the absolute magnitude `amag` and standard error of the parallax `e_plx` of the stars, and the absolute magnitude `amag` and visual apparent magnitude `vmag` of the stars.

Data can also be validated using an external library. Hopsworks can be integrated with the data validation tool Great Expectations as part of the feature pipeline. The validation reports generated by Great Expectations can be stored in Hopsworks to have a validation history for a feature group. Alerts to email or slack can also be configured for when data ingestion succeeds, fails, or has warnings.

5.3 Comparison

Feast and Hopsworks have multiple similarities, but also differences. Table 5.1 summarises the functionalities of Feast and Hopsworks with regard to the explainability-related concerns listed in Chapter 3. Both Feast and Hopsworks have a similar concept of grouping

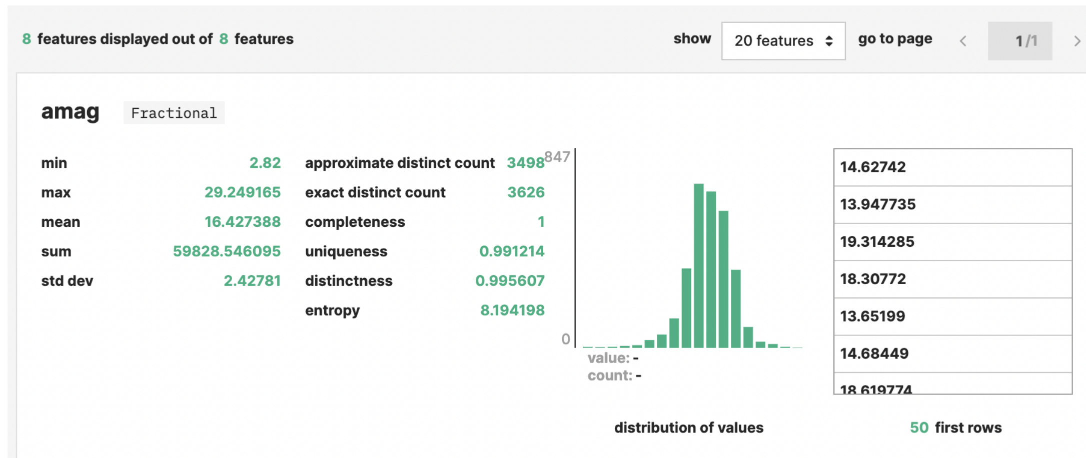


Figure 5.8: Feature statistics in Hopsworks for a numerical feature, with all the statistics enabled. A histogram of the distribution of feature values and the first rows of feature values are also shown.

related features. In Feast, they are called feature views, and in Hopsworks feature groups. Hopsworks has a database of its own for storing the features and also supports external feature groups where the data is stored outside of Hopsworks. Feast does not include a database, instead, all the features are stored in external databases or files.

Feast's feature services and Hopsworks' feature views are also similar concepts, allowing for combining features from different data sources and keeping track of where those features are used and by whom.

In addition to Hopsworks being a part of a more general MLOps platform, the Hopsworks feature store has more functionalities than the lightweight Feast. Feature transformations, statistics and versioning are functionalities that the Hopsworks feature store offers, but Feast does not. Feast's free-form tags could be leveraged to store all kinds of metadata, but they are not automatic nor is their use required. Feast is not designed to cater to those needs, but other services could be used to complement Feast if needed.

Hopsworks offers more functionalities for improving the explainability of the data than Feast on its own. Complementing services would need to be used with Feast to get similar benefits for feature explainability.

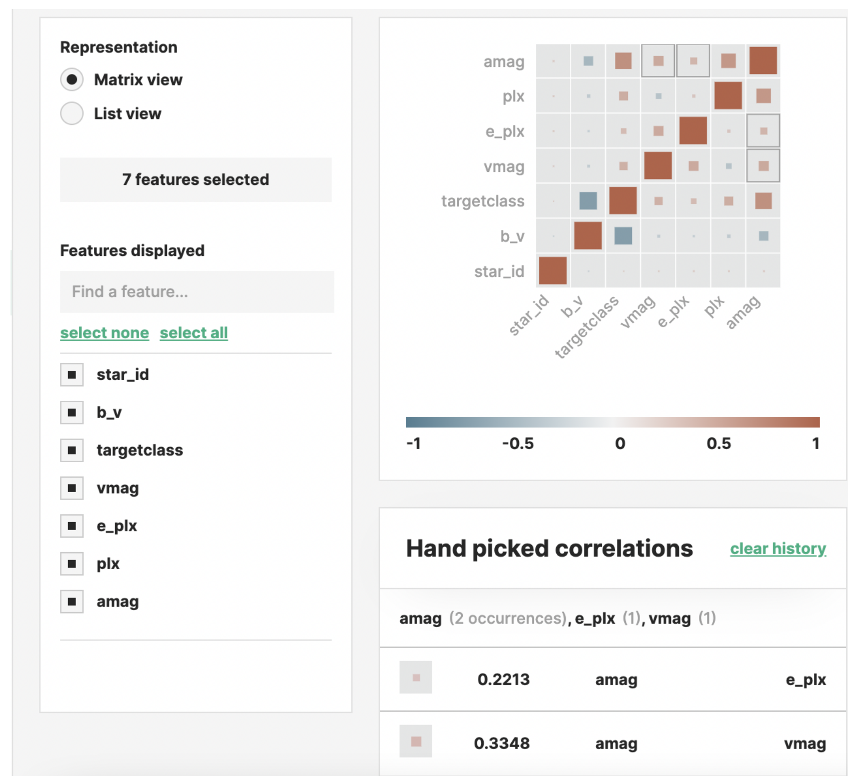


Figure 5.9: Enabling correlations for a feature group results in a correlation matrix for selected features. Precise correlation values are shown for hand-picked correlations or by selecting list view for the feature correlations.

Table 5.1: Summary of the comparison of Feast and Hopsworks in relation to the questions presented at the beginning of Chapter 3.

XAI concerns	**Feast**	**Hopsworks**
Feature definitions	Features have a name, data type, and tags defined.	Features have a name, description, and data type defined.
Standardised metadata	Any pairs of key-value strings as metadata for features, feature views, and feature services.	Feature groups have custom tags and keywords, individual features do not.
Tracking transformations	Transformations not tracked automatically. Tags can be used to manually store information on them.	Transformations done inside Hopsworks are tracked automatically.
Combining multiple sources	Feature services can combine multiple feature views.	Feature views can consist of multiple feature groups.
Tracking feature use	Feature services can be used to track where features are used.	Feature views are used to fetch data, and they are tracked in the feature group, as well as who created the feature view.
Feature versioning	No versioning in Feast itself.	Feature groups and transformations are versioned.
Data validation and monitoring	Experimental support for Great Expectations for data validation and quality monitoring. Only for offline features.	Basic statistics by default. Feature distribution, feature correlations, and more exact statistics can be enabled. Supports Great Expectations for data validation.

6 Discussion

In this thesis, we aimed to find out how feature stores can be used in explaining the features used by machine learning models, and how we can use existing tools to achieve this. We formulated the research question: how can feature stores help explain the features used for machine learning models?

In order to synthesise the characteristics that make a feature store, and how they help with understanding the data, we reviewed literature on feature stores and documentation of different existing feature stores. The feature store is used to transform the data into features, store and manage the features, and serve the features for model training and inference. It is noteworthy, that feature stores are still a relatively new subject: Google Scholar gives about 900 hits for the query "feature store", ACM Digital Library 53, IEEE Xplore 5, Scopus 37, and Web of Science 11.

Feature stores can support different explainability methods by connecting the features used by machine learning models to the original data. For example, methods like LIME (Ribeiro et al., 2016) discover the features that were important for a particular prediction, and feature stores can be used to give insight into those features. When explaining for discovery, machine learning models might find new patterns in the features they use for inference, and feature stores can then connect the transformed features to the original data. When explaining for control, feature stores can help by enabling the monitoring of features and data. When using XAI to improve machine learning models, feature stores can help with feature selection by making it easier to find relevant features and by showing statistics on, e.g., the correlation of different feature values.

We compared two different feature stores from an explainability point of view and found that they have significant differences. Despite their differences, the compared feature stores also had much in common, and both were useful for explaining the features.

We limited the comparison of existing feature stores to two open source platforms. However, there are a number of existing tools that were excluded from the comparison to limit the scope of this thesis project and to keep the focus on open source tools. Some notable feature stores excluded from this comparison are Google's Vertex AI Feature Store (Google, 2023), Amazon SageMaker Feature Store (AWS, 2023), and Tecton Feature Platform (Tecton, 2023). The other feature stores might have different approaches and functionalities

than the ones included in the comparison, so they cannot be generalised directly to other feature stores. However, there are some common characteristics that make up a feature store, detailed in Section 2.2, that most of these tools should share.

Feature stores can be especially helpful for organisations with a higher machine learning maturity. Pipeline-centric organisations that have multiple machine learning models using the same data can benefit greatly from using a feature store to share and document features. Organisations focusing on collecting data or developing their first model might not benefit as much, but using a lightweight feature store might be a good choice when creating the data architecture. With sensitive data or applications that have a considerable societal impact – cases when it is important to really understand the data and features used by the machine learning models – feature stores give important insight into the features used.

Feast is meant to be a very minimal feature store, and it primarily supports timestamped structured data. Some of the functionalities of Feast tested in this thesis project were still experimental and might change as Feast is further developed. The experimental functionalities of Feast included in this thesis are the web UI, saved datasets, data validation, on-demand transformations, and direct data ingestion from streaming sources. If the user mainly needs a simple solution for storing and serving timestamped structured data and is willing to use complementing tools for, e.g., versioning, Feast might be a good choice. Hopsworks is not as lightweight a tool but offers more functionalities for example versioning the feature data, keeping track of transformations, and automatically computing statistics of the features. Hopsworks is also a part of the bigger MLOps platform, where the machine learning models can be managed as well as the feature data they use. Altogether Hopsworks seems a more complete and stable tool.

Further research on feature stores could include comparing other existing feature stores as well as combining Feast with other tools for transforming, monitoring, and versioning data could be tested and compared to tools like Hopsworks. There could also be further research into how to efficiently utilise Feast's highly customisable tags for standardised feature metadata across different teams using the same data. In this thesis project, we only tested the feature stores as stand-alone tools. Feature stores are only one part of an MLOps pipeline, and integrating them into a more extensive MLOps pipeline to develop machine learning systems would give us a better understanding of the feature stores' benefits.

7 Conclusions

Feature stores provide tools to help discover and explain features for machine learning models. They provide definitions for features and can be used to attach standardised metadata for understanding the features better. Explanation of the features can help us understand the decisions made by the models, and as a part of an MLOps pipeline give us insight into maintaining and further developing them. Using feature stores can be especially beneficial for organisations with high machine learning maturity, sensitive data, or machine learning models with applications that have a societal impact.

We compared two open source feature stores, Feast and Hopsworks, to explore how they can be used to help explain features used by machine learning models. The feature stores can help understand where the features come from and how they have been transformed – although there were differences in the compared feature stores Hopsworks and Feast. Hopsworks offers more tools for explaining and governing the data and features than Feast, which is more focused on being a lightweight tool for managing and serving features consistently to machine learning models. On the other hand, Feast's custom tags for features can be used to save any kind of metadata to features, whereas Hopsworks' feature definitions are more rigid – only the name, description, and data type are defined in addition to automatically keeping track of transformations done inside of Hopsworks. Feast was recommended if the user needs a simple solution for managing and serving features, and is willing to use complementing services and conventions for things like versioning and tracking transformations. If the user needs a full solution for a feature store, and possibly also a platform for managing machine learning models, Hopsworks is a better choice.

Bibliography

Adadi, A. and Berrada, M. (2018). "Peeking Inside the Black-Box: A Survey on Explainable Artificial Intelligence (XAI)". In: *IEEE Access* 6, pp. 52138–52160. DOI: 10.1109/ACCESS.2018.2870052.

AWS (2023). *Amazon SageMaker Feature Store*. https://docs.aws.amazon.com/sagemaker/latest/dg/feature-store.html. Accessed: 2023-02-10.

Bender, E. M. and Friedman, B. (2018). "Data Statements for Natural Language Processing: Toward Mitigating System Bias and Enabling Better Science". In: *Transactions of the Association for Computational Linguistics* 6, pp. 587–604. DOI: 10.1162/tacl_a_00041.

Bogner, J., Verdecchia, R., and Gerostathopoulos, I. (2021). "Characterizing Technical Debt and Antipatterns in AI-Based Systems: A Systematic Mapping Study". In: *2021 IEEE/ACM International Conference on Technical Debt (TechDebt)*, pp. 64–73. DOI: 10.1109/TechDebt52882.2021.00016.

Del Balso, M. (2020). *What is a Feature Store?* https://www.tecton.ai/blog/what-is-a-feature-store/. Accessed: 2022-10-19.

Diakopoulos, N., Friedler, S., Arenas, M., Barocas, S., Hay, M., Howe, B., Jagadish, H. V., Unsworth, K., Sahuguet, A., Venkatasubramanian, S., Wilson, C., Yu, C., and Zevenbergen, B. (2016). *Principles for Accountable Algorithms and a Social Impact Statement for Algorithms*. fatml.org/resources/principles-for-accountable-algorithms. Accessed: 2022-10-19.

Dowling, J. (2018). *Logical Clocks raises Seed Funding*. https://www.logicalclocks.com/blog/logical-clocks-raises-seed-funding. Accessed: 2022-11-23.

Feast (2022). *Feast Documentation v0.26*. https://docs.feast.dev/. Accessed: 2022-11-26.

Floridi, L. and Chiriatti, M. (Dec. 2020). "GPT-3: Its Nature, Scope, Limits, and Consequences". In: *Minds and Machines* 30.4, pp. 681–694. ISSN: 1572-8641. DOI: 10.1007/s11023-020-09548-1.

Gebru, T., Morgenstern, J., Vecchione, B., Vaughan, J. W., Wallach, H., III, H. D., and Crawford, K. (Nov. 2021). "Datasheets for Datasets". In: *Commun. ACM* 64.12, pp. 86–92. ISSN: 0001-0782. DOI: 10.1145/3458723.

Google (2023). *Vertex AI Feature Store*. https://cloud.google.com/vertex-ai/docs/featurestore. Accessed: 2023-02-10.

Great Expectations (2023). *Great Expectations Documentation*. https://docs.greatexpectations.io/docs/. Accessed: 2023-01-11.

Guidotti, R., Monreale, A., Ruggieri, S., Turini, F., Giannotti, F., and Pedreschi, D. (Aug. 2018). "A Survey of Methods for Explaining Black Box Models". In: *ACM Comput. Surv.* 51.5. ISSN: 0360-0300. DOI: 10.1145/3236009.

Gunning, D. and Aha, D. (July 2019). "DARPA's Explainable Artificial Intelligence (XAI) Program". In: *AI Magazine* 40.2, pp. 44–58. DOI: 10.1609/aimag.v40i2.2850.

Gunning, D., Vorm, E., Wang, J. Y., and Turek, M. (2021). "DARPA's explainable AI (XAI) program: A retrospective". In: *Applied AI Letters* 2.4, e61. DOI: https://doi.org/10.1002/ail2.61.

Hermann, J. and Del Balso, M. (2017). *Meet Michelangelo: Uber's Machine Learning Platform*. https://www.uber.com/en-FI/blog/michelangelo-machine-learning-platform/. Accessed: 2022-09-05.

ISO (Aug. 2022). *Information technology — DevOps — Building reliable and secure systems including application build, package and deployment*. Standard. International Organization for Standardization.

John, M. M., Olsson, H. H., and Bosch, J. (2021). "Towards MLOps: A Framework and Maturity Model". In: *2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pp. 1–8. DOI: 10.1109/SEAA53835.2021.00050.

Ku, W.-F. (July 2020). *Star Categorization Giants And Dwarfs Dataset, vinesmsuic*. https://www.kaggle.com/datasets/vinesmsuic/star-categorization-giants-and-dwarfs.

Logical Clocks (2022). *Hopsworks Documentation v3.0*. https://docs.hopsworks.ai/3.0/. Accessed: 2022-11-28.

Lum, K. and Isaac, W. (Oct. 2016). "To predict and serve?" In: *Significance* 13, pp. 14–19. DOI: 10.1111/j.1740-9713.2016.00960.x.

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

Ormenisan, A. A., Meister, M., Buso, F., Andersson, R., Haridi, S., and Dowling, J. (July 2020). "Time Travel and Provenance for Machine Learning Pipelines". In: *2020 USENIX*

Conference on Operational Machine Learning (OpML 20). USENIX Association. URL: https://www.usenix.org/conference/opml20/presentation/ormenisan.

Orr, L., Sanyal, A., Ling, X., Goel, K., and Leszczynski, M. (July 2021). "Managing ML Pipelines: Feature Stores and the Coming Wave of Embedding Ecosystems". In: *Proc. VLDB Endow.* 14.12, pp. 3178–3181. ISSN: 2150-8097. DOI: 10.14778/3476311.3476402.

Perry, W. L., McInnis, B., Price, C. C., Smith, S., and Hollywood, J. S. (2013). *Predictive Policing: The Role of Crime Forecasting in Law Enforcement Operations*. Santa Monica, CA: RAND Corporation. DOI: 10.7249/RR233.

Pushkarna, M., Zaldivar, A., and Kjartansson, O. (2022). "Data Cards: Purposeful and Transparent Dataset Documentation for Responsible AI". In: *2022 ACM Conference on Fairness, Accountability, and Transparency*. FAccT '22. Seoul, Republic of Korea: Association for Computing Machinery, pp. 1776–1826. ISBN: 9781450393522. DOI: 10.1145/3531146.3533231.

Ribeiro, M. T., Singh, S., and Guestrin, C. (2016). ""Why Should I Trust You?": Explaining the Predictions of Any Classifier". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '16. San Francisco, California, USA: Association for Computing Machinery, pp. 1135–1144. ISBN: 9781450342322. DOI: 10.1145/2939672.2939778.

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: *Advances in Neural Information Processing Systems*. Ed. by C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett. Vol. 28. Curran Associates, Inc. URL: https://proceedings.neurips.cc/paper/2015/file/86df7dcfd896fcaf2674f757a2463eba-Paper.pdf.

Sell, T. and Pienaar, W. (2019). *Introducing Feast: an open source feature store for machine learning*. https://cloud.google.com/blog/products/ai-machine-learning/introducing-feast-an-open-source-feature-store-for-machine-learning. Accessed: 2022-11-23.

Stafford, G. (2020). *Environmental Sensor Telemetry Data*. https://www.kaggle.com/datasets/garystafford/environmental-sensor-data-132k.

Tecton (2023). *Tecton documentation*. https://docs.tecton.ai/. Accessed: 2023-02-10.