



ExtremeEarth

H2020 - 825258

Deliverable

D1.8

Hops data platform support for EO data -version II

Responsible Partner: LC

**Theofilos Kakantousis, Jim Dowling, Vladimir Vlassov, Sina Sheikholeslami, Tianze Wang
Desta Haileselassie Hagos**

Status: **Final**

Scheduled Delivery Date: **30/06/2021**

Executive Summary

The ExtremeEarth project aims at advancing the state of the art in Big Data Analytics for Earth Observation data with Copernicus data. New techniques in the areas of Remote Sensing and Artificial Intelligence with an emphasis on Deep Learning will be developed and used during the course of this project. These techniques will be demonstrated in two use cases, namely, Food Security and Polar Ice. One of the main factors that differentiate this project in comparison to other Earth Analytics ones, is the use of Hops data platform (Hopsworks), a horizontally scalable full-stack Big Data and Artificial Intelligence (AI) platform. Hopsworks provides first-class support for both Data Analytics at scale and Data Science at scale. In particular, Hopsworks supports the development of Deep Learning applications in notebooks and the operation of workflows to support those applications, including data processing at scale, model training at scale, and model deployment. In this deliverable, we describe and demonstrate the services and features of Hopsworks that provide users with the means of building scalable Deep Learning pipelines for Earth Observation (EO) Data, as well as support for discovery and search for EO metadata. Users of the platform typically are data engineers working with data collection and transformation, and data scientists working with training samples and model training. This deliverable serves as a demonstrator and walkthrough of the stages of building a production level model that includes Data Ingestion, Data Preparation, Feature Extraction, Model training, Model Serving, and Monitoring. We provide a practical example that demonstrates the aforementioned stages with real-world EO data and provides source code that implements the functionality in the platform. Work contributed to this deliverable will be used by WP2 and WP3.

Updates in D1.8 compared to D1.4

This deliverable builds on D1.4 which is the first version of this deliverable. It delivers the previous content with updates and additions across all sections. In addition, the demonstrator has been updated and is available online [18]. Notable changes compared to the previous version of this deliverable are the following:

1. Section 3.3 Python environment: New section that demonstrates the Python environment support in Hopsworks.
2. Section 4 *Metadata support for EO and Big Data*: Updated section to include implementation details of how Hopsworks has been extended with metadata support for EO data. New sections 4.1 and 4.2 demonstrate how to use metadata to perform free-text search and tagging.
3. Section 5 *EO data pre-processing*: Demonstrates new tools Hopsworks has been extended with for EO data pre-processing.
4. Section 7.2 *Feature validation*: Updated section with new feature validation framework.

-
5. Section 12 *Hopsworks TEPs Integration*: Content was moved and expanded in deliverable D1.6 Platform Software Architecture version II.
 6. Demonstrator code is now available at <https://github.com/ExtremeEarth-Project/eo-ml-examples>






Document Information

Contract Number	H2020 - 825258	Acronym	ExtremeEarth
Full title	From Copernicus Big Data to Extreme Earth Analytics		
Project URL	http://earthanalytics.eu/		
EU Project officer	Riku Leppänen		

Deliverable	Number	D1.8	Name	Hops data platform support for EO data version II	
Task	Number	T1.4	Name	EO Data Pipelines	
Work package	Number	WP1			
Date of delivery	Contractual		M30	Actual	M30
Status	draft <input type="checkbox"/> final <input checked="" type="checkbox"/>				
Nature	Prototype <input checked="" type="checkbox"/> Report <input checked="" type="checkbox"/>				
Distribution Type	Public <input checked="" type="checkbox"/> Restricted <input type="checkbox"/> Consortium <input type="checkbox"/>				
Responsible Partner	LC				
QA Partner	KTH, UoA, NCSR-D				
Contact Person	Theofilos Kakantousis				
	Email	theo@logicalclocks.com	Phone		Fax

Project Information

This document is part of a research project funded by the IST Programme of the Commission of the European Communities as project number H2020-825258. The beneficiaries in this project are the following:

Partner	Acronym	Contact
National and Kapodistrian University of Athens Department of Informatics and Telecommunications (Coordinator)	UoA 	Prof. Manolis Koubarakis National and Kapodistrian University of Athens Dept. of Informatics and Telecommunications Panepistimiopolis, Ilissia, GR-15784 Athens, Greece Email: koubarak@di.uoa.gr Tel: +30 210 7275213, Fax: +30 210 7275214
VISTA Geowissenschaftliche Fernerkundung GmbH	VISTA 	Heike Bach Email: bach@vista-geo.de
The Arctic University of Norway Department of Physics and Technology	UiT 	Torbjørn Eltoft Email: torbjorn.eltoft@uit.no
University of Trento Department of Information Engineering and Computer Science	UNITN  UNIVERSITY OF TRENTO - Italy Department of Information Engineering and Computer Science	Lorenzo Bruzzone Email: lorenzo.bruzzone@unitn.it
Royal Institute of Technology	KTH 	Vladimir Vlassov Email: vladv@kth.se







National Center for Scientific Research - Demokritos	NCSR-D  DEMOKRITOS	Vangelis Karkaletsis Email: vangelis@iit.demokritos.gr
Deutsches Zentrum für Luft-und Raumfahrt e. V.	DLR  DLR	Corneliu Octavian Dumitru Email: corneliu.dumitru@dlr.de
Polar View Earth Observation Ltd.	PolarView  Polar View <small>Earth Observation for Polar Monitoring</small>	David Arthurs Email: david.arthurs@polarview.org
METEOROLOGISK INSTITUTT	METNO  Norwegian Meteorological Institute	Nick Hughes Email: nick.hughes@met.no
Logical Clocks AB	LC  LOGICAL CLOCKS	Jim Dowling Email: jim@logicalclocks.com
United Kingdom Research and Innovation – British Antarctic Survey	UKRI-BAS  British Antarctic Survey <small>NATURAL ENVIRONMENT RESEARCH COUNCIL</small>	Andrew Fleming Email: ahf@bas.ac.uk

Table of Contents

1. EO Data Deep Learning Pipelines Architecture	9
2. Parallel Data Processing with Apache Spark	12
3. Development Environment	18
3.1 Jupyter notebooks on Kubernetes	18
3.2 Jupyter notebooks as Jobs	19
3.3 Python environment	20
4. Metadata support for EO and Big Data	22
4.1 Search example	24
4.2 Tagging example	25
5. Data Ingestion	28
5.1. Demonstrator Dataset	28
5.2. Accessing EO Data from Hopsworks	31
6. EO data pre-processing	33
6.1 EO data pre-processing with Python	33
6.2 EO data pre-processing with Docker and Kubernetes	34
7. Feature Engineering and Data Validation	38
7.1 Feature Store	38
7.2 Feature Validation	43
8. Training	48
8.1 Experiments	48
8.2 Parallel Experiments	52
8.3 Distributed Training	55
8.4 Hyperparameter Tuning with Maggy	56
8.5 Ablation studies	61
9. Model Analysis	64
10. Model Serving & Monitoring	66
11. Orchestration	73
12. Hopsworks TEPs Integration	75

13. References

76

1. EO Data Deep Learning Pipelines Architecture

A Data Science application, particularly in the domain of Big Data, typically consists of a set of stages that form a data pipeline. This data pipeline is responsible for transforming data and serving it as knowledge by using data engineering processes and by applying ML algorithms and Deep Learning (DL) techniques. These stages typically are:

- Data Ingestion
- Data Preparation & Validation
- Feature Extraction
- Build & validate the model (training)
- Model Serving/Monitoring

The first two steps, data ingestion, and preparation can also be described as Data Pipelines. Figure 1 illustrates the Deep Learning pipeline stages along with the stakeholders of these stages. Feature Extraction is facilitated by the Feature Store service, presented in section 7.

HORIZONTAL SCALABILITY AT EVERY STAGE IN THE PIPELINE

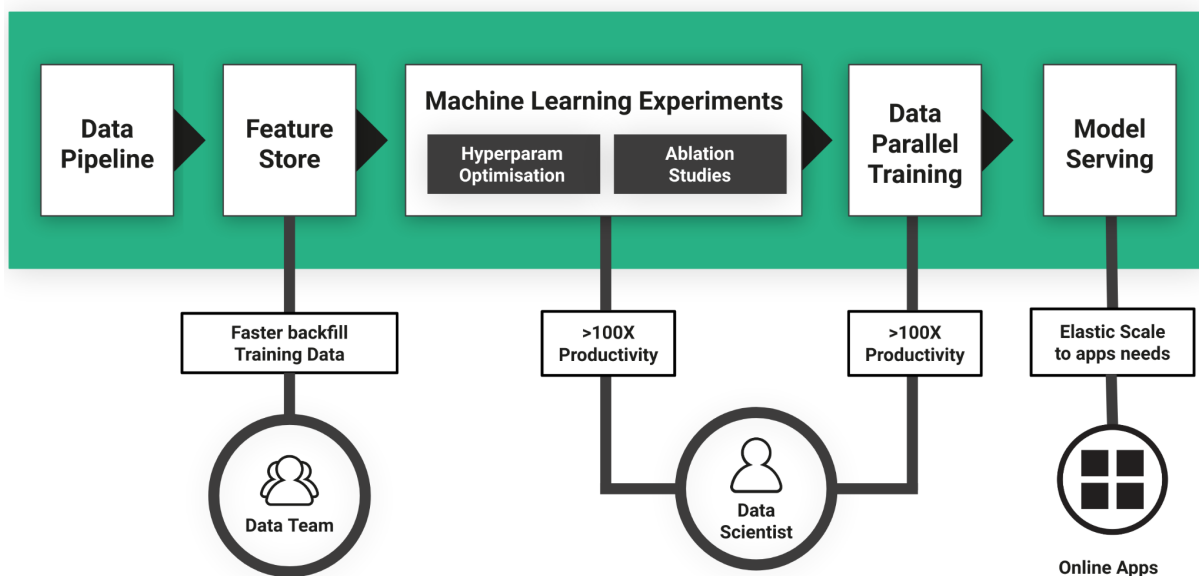


Figure 1. Machine Learning Pipeline [1]

Once data goes through all the stages and the model of the ML pipeline is served, the stages are executed once again in order to consider new data that has arrived in the meantime and to further fine-tune the pipeline stages which will lead to more accurate models and results. Therefore, a data science life cycle is formed, which continuously iterates the above ML pipeline.

As a result, data scientists are faced with the highly complex task of developing DL workflows that utilize each stage of the ML pipeline. The complexity of such pipelines can

grow as the input data increases in volume, which in the case of EO data means that a robust and flexible architecture needs to be in place to assist data engineers and scientists to develop these pipelines. Figure 2 depicts the overall architecture and lifecycle of a DL pipeline along with the technologies used to implement it and demonstrated in the rest of this document.

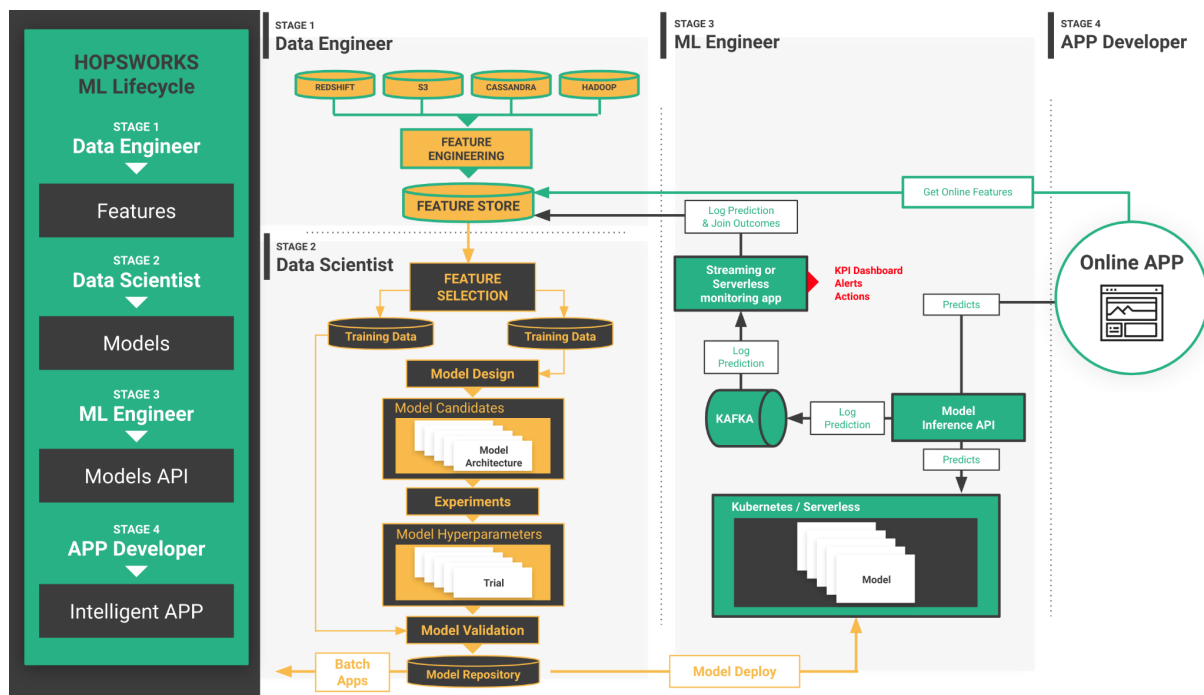


Figure 2. Hopsworks ML Lifecycle

The stages of data collection (ingestion), pre-processing, and management of a service to store curated feature data and compute features can be considered to be a part of the Data Engineering lifecycle. The Feature Store is the service used in this DL pipeline to manage curated feature data. The second step of the pipeline, the actual ML training and model development, starts by fetching feature data in appropriate file formats to be used as input for training, with the file format depending on the ML framework that is used. This step can be considered as the Data Science lifecycle, where new feature data is fetched and new models are iteratively developed and pushed to production (serving).

One of the main goals of a DL pipeline is to continuously improve the output models, by using some user-defined metrics. To detect when the DL pipeline should be triggered in order to update a DL model served in production, there needs to be a mechanism in place that logs all inference requests and monitors how the model is performing over time. Model serving and monitoring in Hopsworks provide these capabilities to developers of pipelines and are further demonstrated in section Model Serving & Monitoring.

Figure 3 depicts the Hopsworks services stack. HopsFS and RonDB (NDB/MySQL Cluster) provide horizontally scalable data and metadata storage. Apache Hadoop YARN and Kubernetes are the resource management frameworks on the upper layer. These provide resources to the distributed processing framework in Hopsworks, Apache Spark, and to Hopsworks itself to provide EO data pre-processing with arbitrary programming languages functionality and also running Python jobs and notebooks. Auxiliary services are part of this layer, providing logging and metrics monitoring. The next layer comprises Hopsworks itself, the webapp with the REST API that provides client applications and users connectivity to the entire Hopsworks cluster.

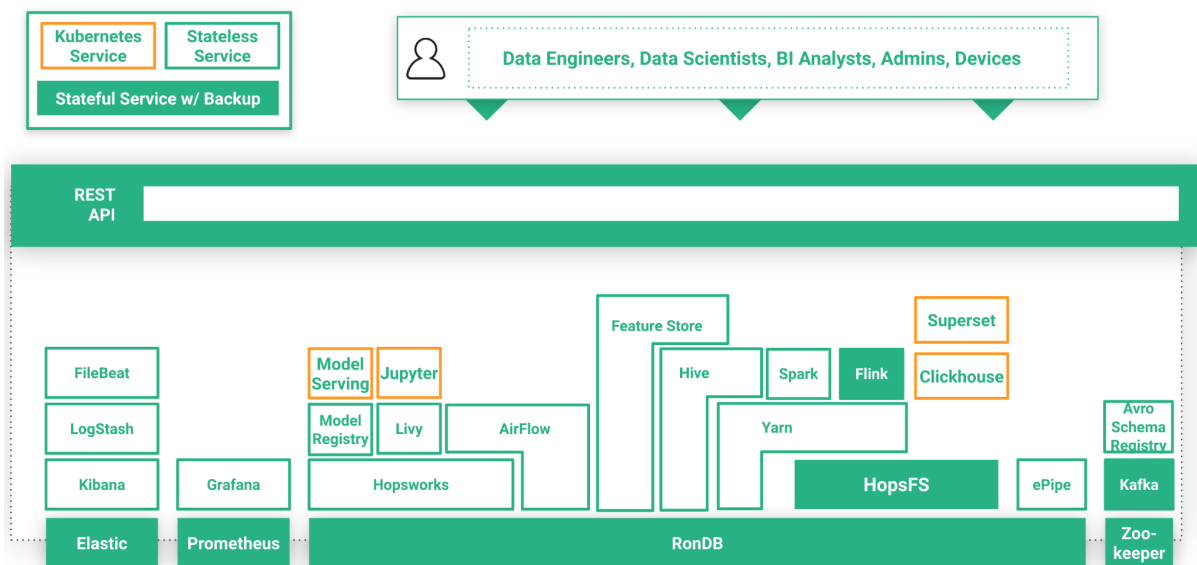


Figure 3. Hopsworks services stack

Running a DL pipeline can be a repetitive task, as most (if not all) stages need to run when new data is ingested into the system. Orchestrating the order of stage execution, monitoring progress, and putting a retry mechanism in place in case of failures, is an important part of making an EO data pipeline production ready. Hopsworks integrates Apache Airflow[2] as an orchestration engine and section 11 describes how this integration has been implemented and provided to users of the platform. The rest of this document demonstrates each stage of the Deep Learning pipeline implemented in Hopsworks, with a dataset based on a real-world dataset from the Polar use case domain.

2. Parallel Data Processing with Apache Spark

Integration Guidelines section of deliverable *D1.5 Hops data platform integration guide for applications - version II* describes in detail how the Jobs and Notebooks services in Hopsworks can be used for data processing and deep learning. This section describes how the aforementioned services utilize Apache Spark in Hopsworks and how the latter is integrated into Hopsworks to provide horizontally scalable data processing and distribution of machine learning processes in a Hopsworks cluster.

Apache Spark (Spark) is a framework for large-scale distributed processing of data. Spark provides data engineers and data scientists with the tools to process data at scale and offers powerful APIs for developing data pipelines. Adding support for developing and running Apache Spark applications at scale with Hopsworks is also a prerequisite for the development of part of WP3, in particular, deliverables related to “Software for querying and extreme analytics for big linked geospatial data”.

Spark can be deployed on top of various resource management services, such as Apache YARN (YARN), Mesos, and Kubernetes. Hopsworks utilizes the flavor of YARN that is developed within the Hops project as the resource management service for deploying distributed applications on a cluster of servers. YARN in Hops supports scheduling applications with resource constraints, which are CPU, main memory, and Graphics Processing Unit (GPU). Therefore, Spark in Hopsworks is deployed on top of YARN, and users developing DL pipelines can easily, from within the Hopsworks user interface (UI), request these three resources to be allocated to their job or notebook. That is particularly important for allocating GPUs when the Spark program needs to have access to GPU compute power.

During the ExtremeEarth project Spark versions 2.x, in particular version 2.4.3, were used for the majority of the duration of the project, which do not support scheduling with GPUs [11], unlike recently released Spark 3.x versions that added such support. Therefore, Spark was extended within the scope of the ExtremeEarth project to provide GPU-based resource allocation and scheduling within the Hopsworks platform. In Hopsworks, a new parameter “spark.executor.gpus” has been added to Spark to indicate the number of GPUs to be requested for each Spark executor. It is then left up to the scheduler of YARN to allocate the resources based on scheduler policies and current cluster utilization. Implementation details are available online at the Logical Clocks GitHub repository [41]. Figures 4 and 5 provide a high-level view of the architecture behind GPU and Deep Learning framework support within PySpark.

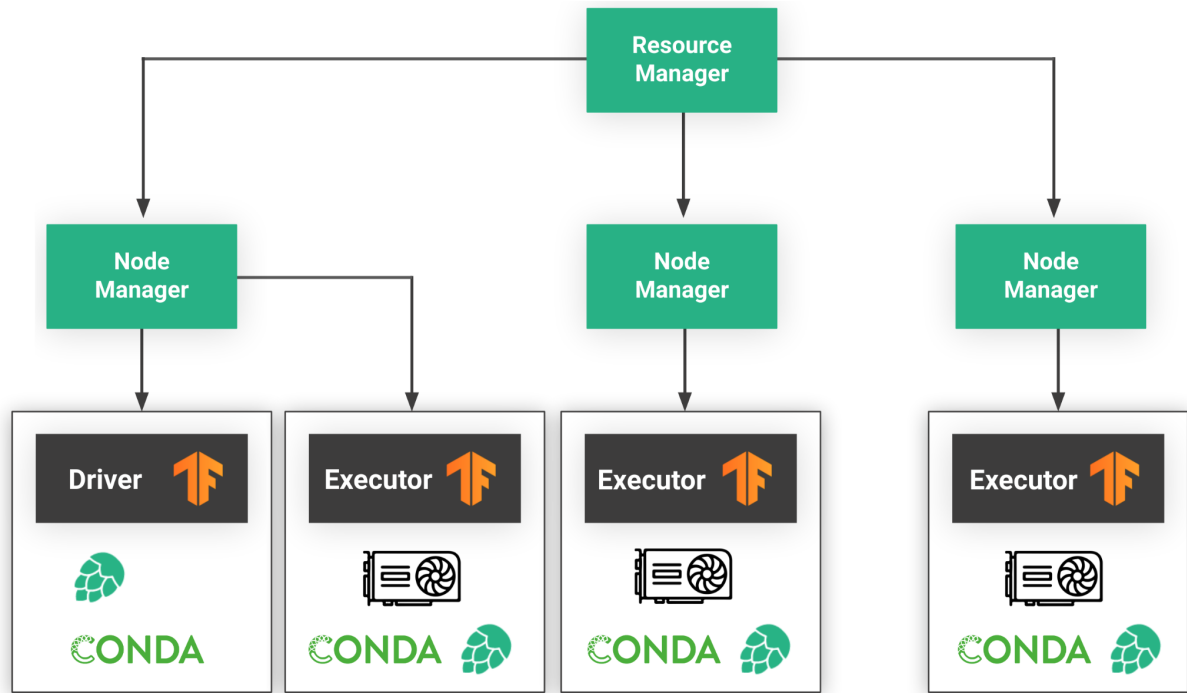


Figure 4. Resource management and GPU allocation in Hopsworks

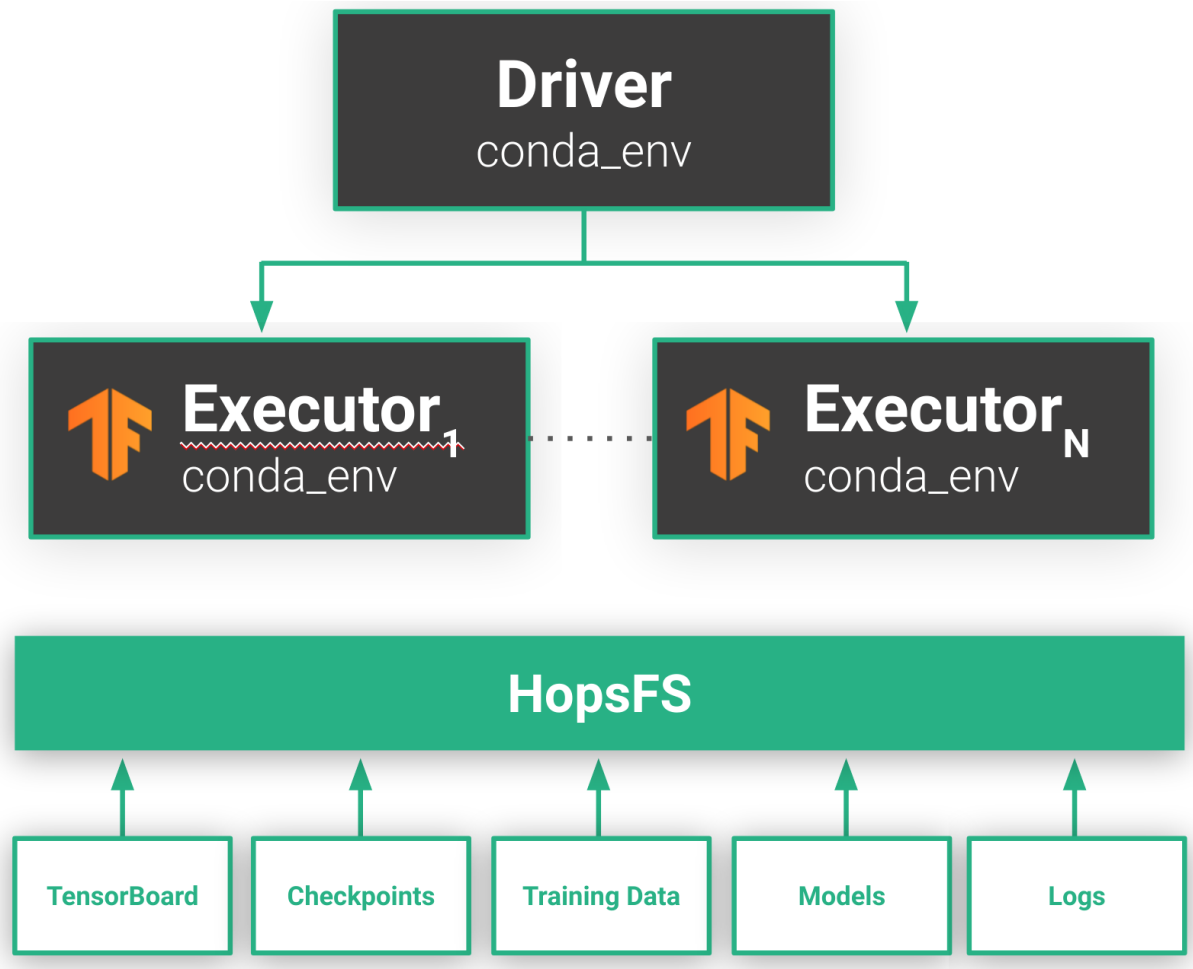


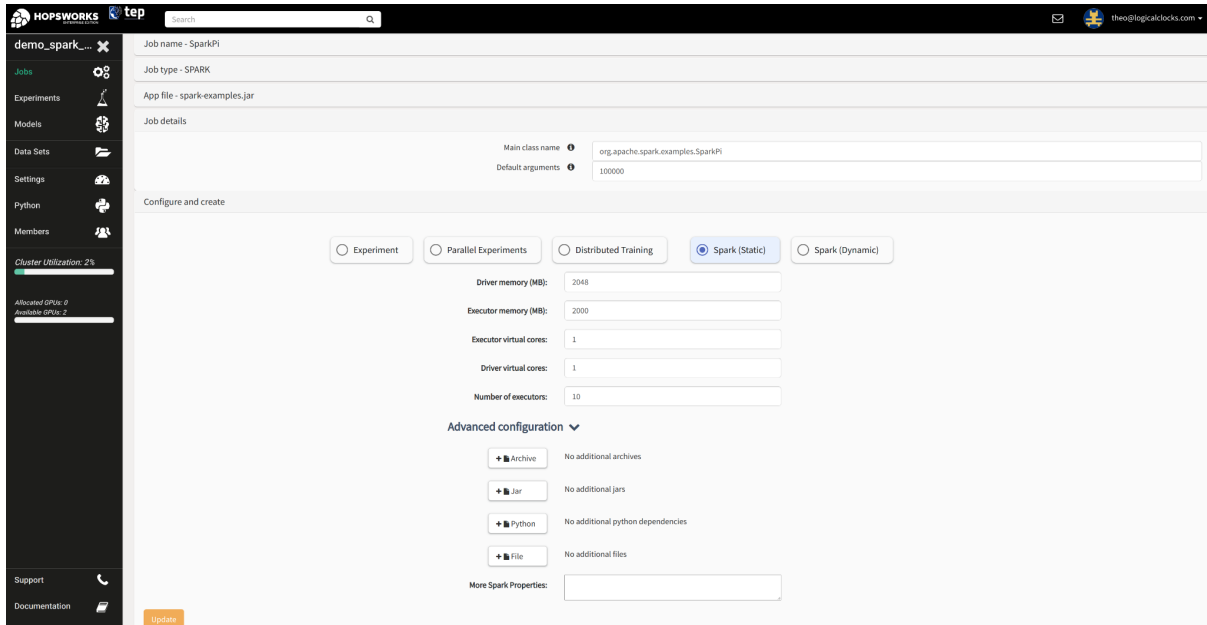
Figure 5. Spark Driver and Executors with GPUs and Deep Learning Frameworks

Hopsworks has been extended with an API that allows clients to easily submit Spark applications on the cluster. Hopsworks sets up default Spark configuration parameters and in addition, it provides a flexible way for users to provide additional ones with their Spark application via the UI or the RESTful and client APIs for their application.

Endpoints for submitting Spark applications as jobs in Hopsworks can be found under the *jobs* resource, which is a sub-resource of the top-level *projects* resource. To submit a new job, a client would need to submit an HTTP request to the Hopsworks `/project/{projectId}/jobs/{name}/executions` endpoint. An exhaustive list of all endpoints related to the Jobs resource is available at SwaggerHub [6].

Figure 6 demonstrates how users can submit a Spark job via UI of the Hopsworks Jobs service with dynamic executors. All configuration parameters are provided through the UI and sent via HTTP to a RESTful jobs service endpoint in Hopsworks. The executable JAR file “spark-examples.jar” has been uploaded into the TestJob dataset and program runtime input arguments are set either with the “Default arguments” textbox or with the job arguments

popup textbox when the user runs the job. All metadata of jobs, such as Spark configuration, duration, creator etc., are stored in the metadata layer of Hopsworks. This metadata layer is backed by the in-memory horizontally scalable distributed database RonDB, an evolution of MySQL Cluster (NDB). This database is used to store metadata of critical Hopswork services, providing strongly consistent metadata across services in Hopsworks.



The screenshot shows the Hopsworks Jobs service interface. The left sidebar contains navigation links: Jobs, Experiments, Models, Data Sets, Settings, Python, and Members. The main content area is titled 'demo_spark_...' and shows the configuration for a Spark job. The job name is 'SparkPi', the job type is 'SPARK', and the app file is 'spark-examples.jar'. The job details section shows the main class name as 'org.apache.spark.examples.SparkPi' and default arguments as '100000'. The 'Configure and create' section has radio buttons for 'Experiment', 'Parallel Experiments', 'Distributed Training', 'Spark (Static)' (selected), and 'Spark (Dynamic)'. Below these are input fields for 'Driver memory (MB): 2048', 'Executor memory (MB): 2000', 'Executor virtual cores: 1', 'Driver virtual cores: 1', and 'Number of executors: 10'. An 'Advanced configuration' section includes buttons for '+ Archive', '+ Jar', '+ Python', and '+ File', each with a 'No additional' status. A 'More Spark Properties' section is also visible at the bottom.

Figure 6. Submitting Spark Applications from the Jobs services

Furthermore, Hopsworks collects real-time logs produced by the Spark driver and executors, by utilizing the ELK stack which comprises Elasticsearch, Logstash, Kibana, and Beats [35]. Filebeat, part of the Beats suite of data shippers, is installed on all cluster nodes that may run either a driver on an executor and it monitors the directory where logs are output. Log content is then transmitted over the network to Logstash which applies some business logic by using Grok filters. The purpose of this business logic is to enrich the logs with metadata in regards to which project particular logs belong to and then logs are sent to Elasticsearch for indexing. Elasticsearch stores all logs of a project in one index. Kibana is provided via the Jobs UI in Hopsworks as a tool to create rich visualizations of logs. Figure 7 shows the logs of a Spark job in Hopsworks. All logging services are adapted to the project-based multi-tenant model of Hopsworks, which means users can securely access logs produced by jobs and notebooks within their projects. As a result, it is not possible to access logs of other projects, preventing data leaks. To achieve that, logging services utilize the project-user TLS certificates mechanism that is also used consistently across Hopsworks to ensure authentication, authorization and encrypted communication [53].

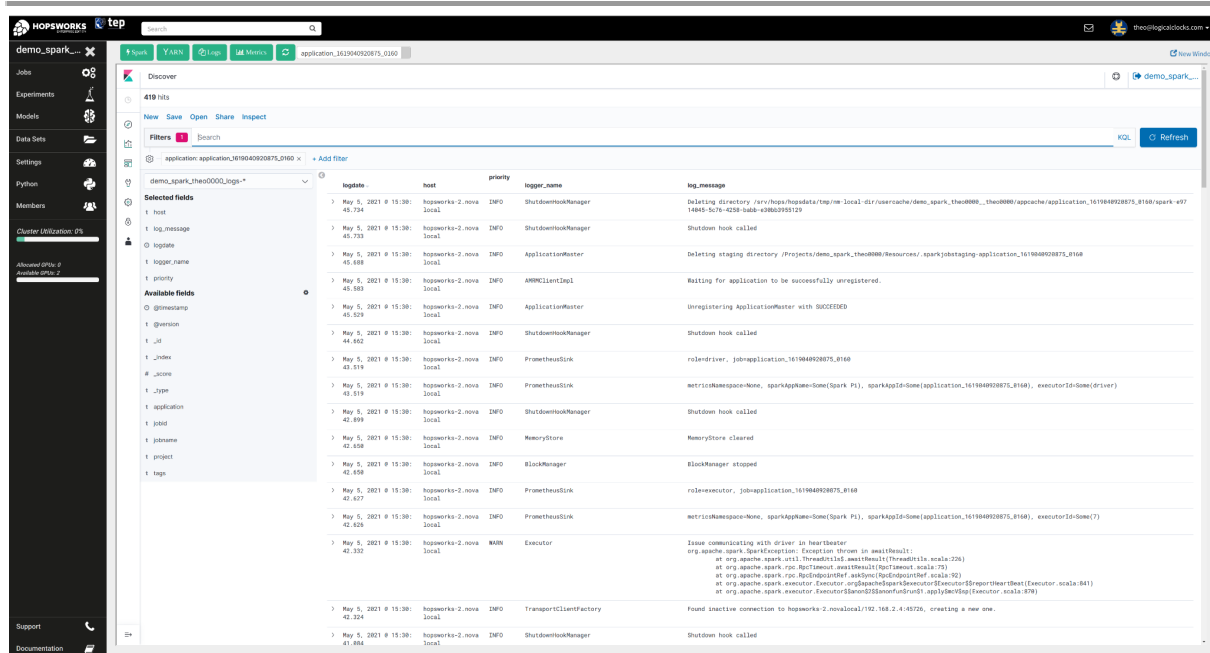


Figure 7. Real-time Apache Spark logs in Hopworks with the Jobs service

In addition, metrics that are reported from Spark are collected with Prometheus [42], an open-source system and service monitoring system. It collects metrics from configured targets at given intervals, evaluates rule expressions, displays the results, and can trigger alerts when specific conditions are observed. Metrics in Hopworks are visualized using Grafana, an open-source analytics, and monitoring framework [43]. The latter can be accessed via the Jobs UI in Hopworks. The previous version of this deliverable, D1.4, used InfluxDB instead of Prometheus for storing job and service metrics. The move to Prometheus was prompted by the wider adoption of the Prometheus system and its more optimal resource utilization as it was observed that InfluxDB was using too many compute resources on the Hopworks cluster. Further information regarding how logging and monitoring can be accessed by users is provided in deliverable 1.1 under section Jobs. Figure 8 shows metrics of Spark job run in Hopworks, with 10 executors.

Again, similarly to the real-time logging architecture described above, access to metrics is based on project-user authorization, meaning users can only access metrics of projects they are members of. This metrics monitoring architecture is also used in Hopworks to collect logs of services at an administrative level. A Hopworks user with the role HOPS_ADMIN, via the “Admin UI”, can monitor utilization in real-time of critical services such as the distributed file system HopsFS and the metadata layer (database).

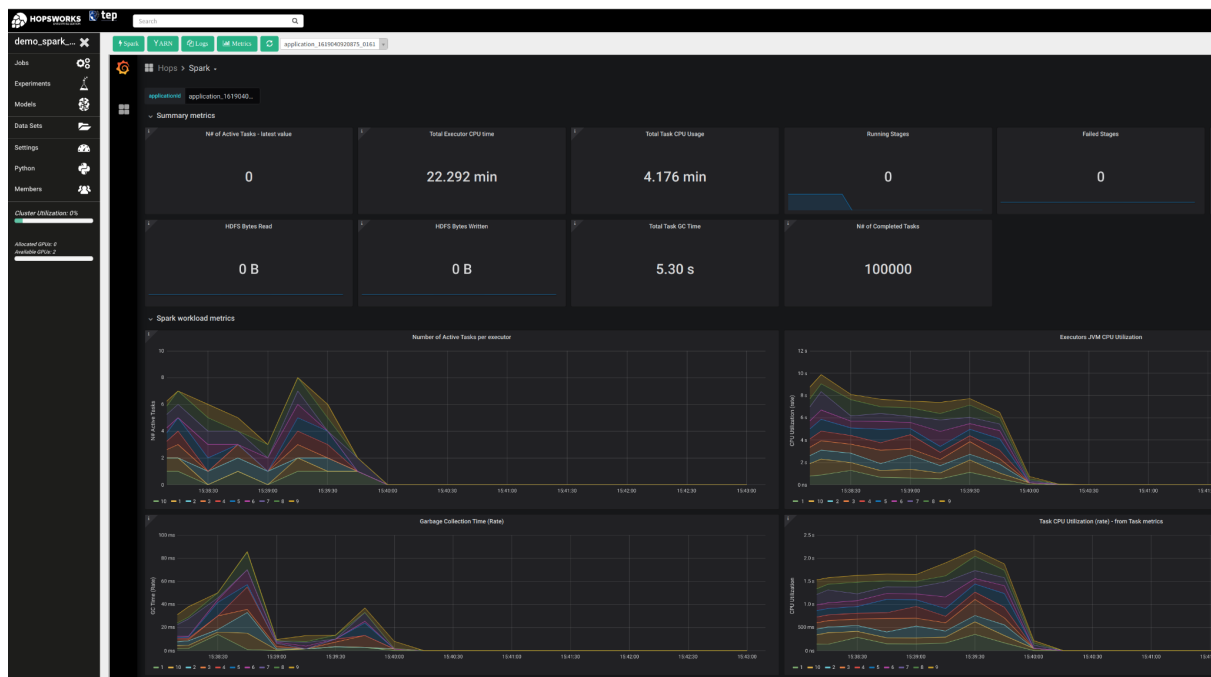


Figure 8. Real-time Job (Spark) metrics monitoring with Prometheus and Grafana in Hopworks

3. Development Environment

Jupyter notebooks have become the lingua franca for data engineers and data scientists that develop both data and deep learning pipelines [44]. They provide an interactive way of writing code, getting and sharing results and code with other developers. Hopsworks supports writing Python and Spark/PySpark programs with Jupyter and JupyterLab via the Jupyter service which is accessible from the navigation bar from within a project in Hopsworks. Users are presented with a set of configuration parameters with some of them already set with default values.

To have a consistent way of working with Jupyter notebooks and jobs in Hopsworks, the Jupyter configuration is common with the Jobs one. That means users can set a Spark configuration to use when using the Jupyter server and that configuration is stored in the Hopsworks database and made available in the Jobs service as well. This reduces user friction, allowing users to reuse configurations across jobs and notebooks allowing for a seamless development experience.

Figure 9 shows the default Jupyter dashboard page in Hopsworks for working with Spark/PySpark notebooks.

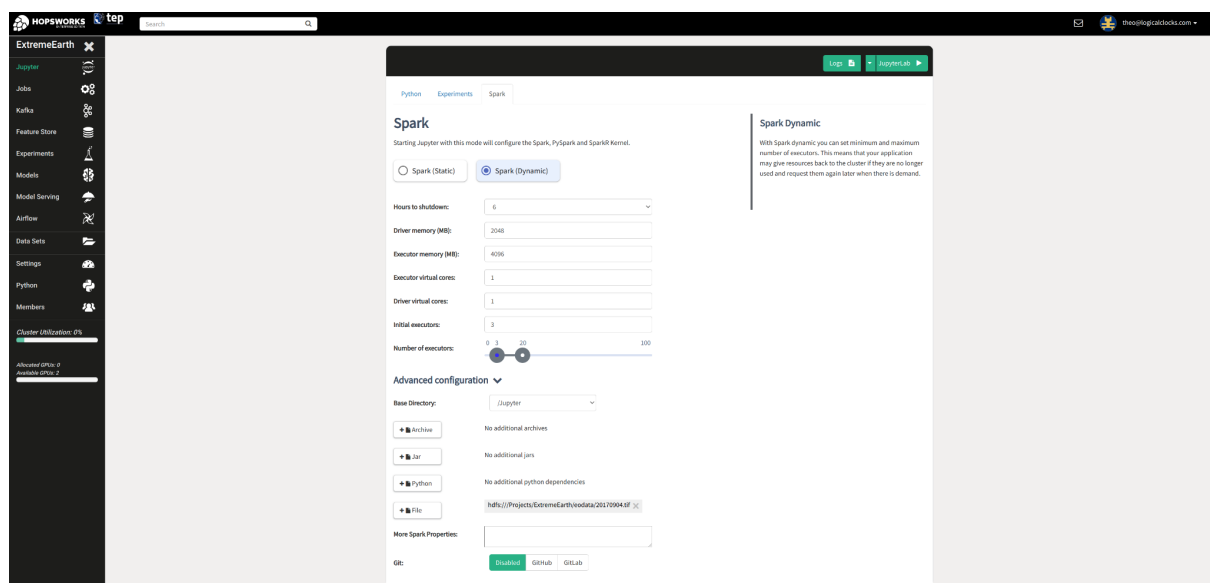


Figure 9. Jupyter dashboard landing page

3.1 Jupyter notebooks on Kubernetes

Hopsworks has been extended to run Jupyter notebooks in Docker containers managed by a Kubernetes cluster. The latter is installed and configured automatically with Chef [28] by using Karamel [36], the installation framework of Hopsworks, during a Hopsworks cluster installation. When users create a new project, a new namespace in Kubernetes will be

created for this project. For each user starting the Jupyter server in Hopsworks, the latter creates a Kubernetes deployment that runs the Jupyter server and a sidecar that collects the logs. These logs are sent to the ELK stack described in section 2 and displayed via Kibana to the users within the Hopsworks UI.

A Docker image is pre-built and provided by Hopsworks that is responsible for running the Jupyter server. The Docker file of this image is based on the official ubuntu one found on Dockerhub and is responsible for the following:

1. Set the system user to run the container.
2. Install all necessary dependencies such as Java OpenJDK8.
3. Expose the port on which the Jupyter server is listening at.
4. Set the base working directories for infrastructure dependencies such as the project's Python environments based on the Anaconda distribution [45].
5. Start the Jupyter server.

In addition, this Docker image already contains a plethora of frameworks and libraries that data scientists working with EO data typically use today. Such frameworks are TensorFlow version 2.4 as per the time of writing, PyTorch version 1.7, cudatoolkit for working with GPUs, and many more. Users can install libraries on top of this image and the architecture of this mechanism is described in section 3.3 Python environment management.

Therefore, Hopsworks connects to Kubernetes to submit deployments and Kubernetes manages the lifecycle of the Jupyter deployment. Hopsworks exposes lifecycle operations to Hopsworks users and manages user authentication via its HTTP RESTful API with operations such as start/stop the Jupyter server.

3.2 Jupyter notebooks as Jobs

Jupyter notebooks in Hopsworks can also run as regular PySpark or Python jobs, as Hopsworks converts automatically the notebook into a PySpark or Python program which is then submitted via the regular Hopsworks Jobs service. Figure 10 depicts the series of events that occur for launching notebooks and jobs in Hopsworks, which is what Airflow also leverages to provide DL pipeline orchestration. In short, Hopsworks reads the input notebook file and converts it to an executable Python script using nbconvert [54]. The PySpark or Python job then reads and executes it as it would with any other regular PySpark or Python job. Hopsworks operators in Airflow can then be used to create a workflow, in the form of a Directed Acyclic Graph (DAG), that makes use of the aforementioned jobs. Furthermore, whenever the notebook is updated by the user, the changes are reflected on a new job execution as the notebook is converted to a Python file upon every execution.

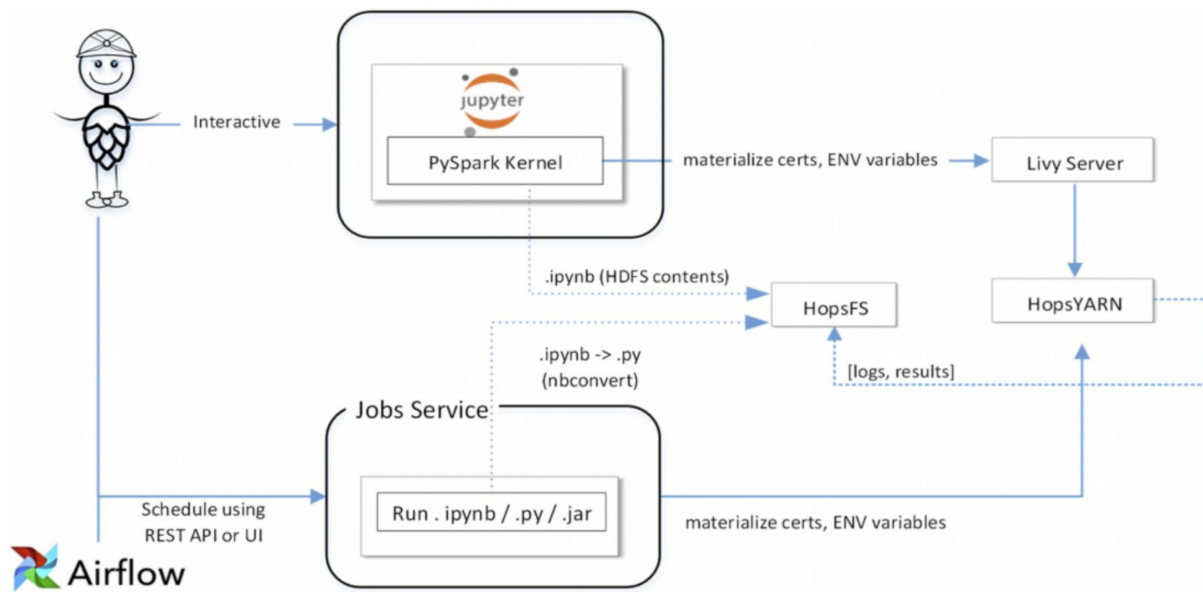


Figure 10. Jobs and notebooks execution framework

3.3 Python environment

Hopsworks has been extended with first-class support for self-service management of Python dependencies within Hopsworks projects. A Hopsworks project is created along with its own Python Anaconda environment [45]. This enables Python dependency isolation between projects which is crucial in having a stable development and job execution environment. The Hopsworks REST API exposes the same functionality to non-UI clients via the PythonResource resource, with source code available at [5] and REST API documentation at [6] under “/project/{projectId}/python/environments”. This Python environment is shipped with the Docker image described in section 3.1. Users in Hopsworks manage this environment through the UI in a self-service manner. Users therefore can:

- Search for Python libraries through pip or conda.
- Install libraries through pip, conda, git, wheel files.
- Uninstall libraries.
- List libraries.
- Export the Python environment in a yml file which can then be imported into another project or used as a backup mechanism.

Python libraries might conflict with each other, resulting in potential incompatibilities when running jobs and notebooks. Hopsworks has been extended with a mechanism that detects such conflicts and displays them to the user. Figure 11 shows the Python environment of a project and an example of such a conflict warning. Users can then take appropriate action by installing the library versions of their choice.

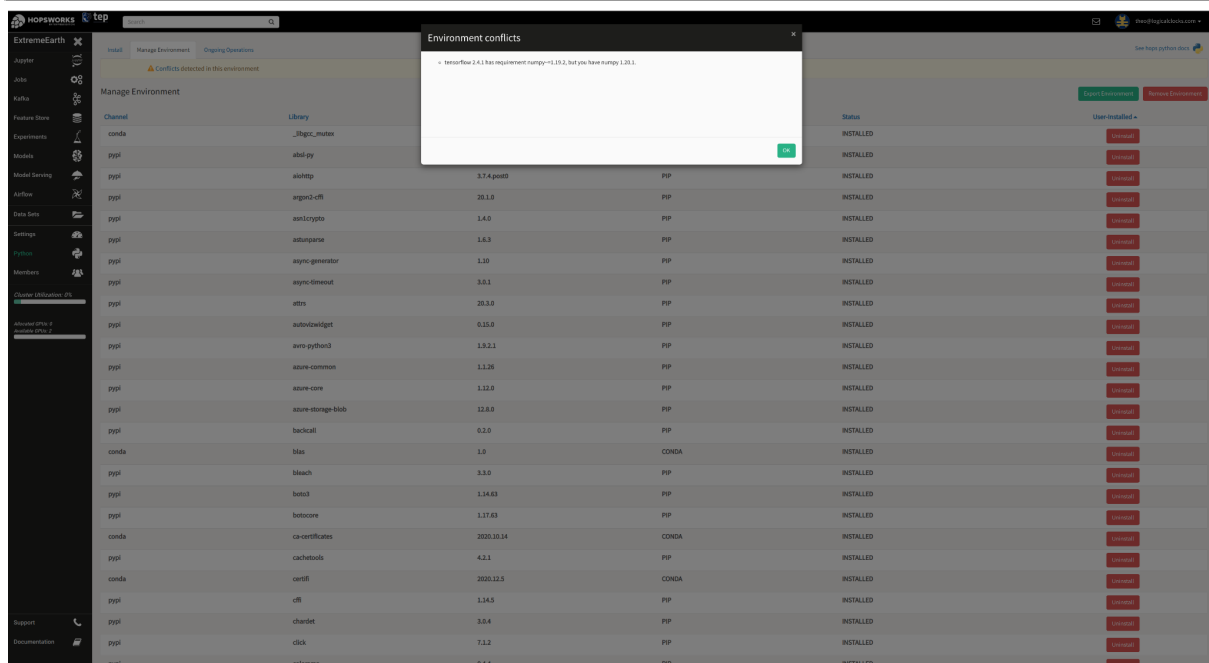


Figure 11. Jobs and notebooks execution framework

Since the Python Anaconda environment is included in the Docker image of the project, subsequent operations such as library install/uninstall are managed by Hopsworks by adding additional layers on top of the project's Docker image. Hopsworks is responsible for the Docker image lifecycle: building the image, pushing it to the registry, and removing it if the Python environment is reset or the project is deleted.

Some libraries that have been utilized by the ExtremeEarth use cases and WP4 "The Food Security Use Case" and WP5 "The Polar Use Case" are TensorFlow for deep learning and GDAL for data pre-processing for which more details are available in section 6 EO data pre-processing.

4. Metadata support for EO and Big Data

Within the Hopsworks platform, data storage is managed by the distributed file system HopsFS. Hopsworks has been extended with metadata management for datasets and files to support both searching for data and tagging data with user-provided metadata. By providing support for metadata, it is ensured that EO data can be properly annotated, cataloged, and made accessible to appropriate users.

Metadata in Hopsworks is used primarily to discover and retrieve relevant files, directories, datasets, and projects through the use of full-text search. Metadata is associated with a particular file and is stored in the same database as the filesystem metadata of HopsFS. Foreign keys in the database link the extended metadata with the target file, ensuring its integrity and consistency. Extended metadata is exported to Elasticsearch, from where it can be queried and the associated file/directory/dataset/project can be discovered and accessed.

Hopsworks has been extended with metadata tagging capabilities for EO data. Tags are additional metadata, extended metadata in the Hopsworks terminology, attached to artifacts in Hopsworks, and thus they can be used not only for an enhanced full-text search but also to provide users with a more dynamic metadata content that can be used for both storages as well as enhancing artifact discoverability [61]. A tag is a *{key: value}* association, providing additional information about the data, such as for example geographic origin. This is useful in an organization as it adds more context to data making it easier to share and discover data and artifacts.

A schema needs to be defined for the tags when they are attached to datasets. Schemas follow <https://json-schema.org> as reference. The schemas define legal jsons and these can be primitives, objects, or arrays. The schemas themselves are also defined as jsons. Allowed primitive types are:

- string
- boolean
- integer
- number (float)

Complex schemas can be defined as well, for example

```
{
  "type" : "object",
  "properties" :
  {
    "location" : { "type" : "string" },
    "dimensions" : { "type" : "string" },
    "coordinates" : {
      "type" : "array",
```

```
    "items" : { "type" : "string" }  
  },  
  "required" : ["location", "dimensions"],  
  "additionalProperties": false  
}
```

and a value for this tag is

```
{  
  "location" : "North Pole",  
  "dimensions" : "12345x6789",  
  "coordinates" : ["90.0000°N, 135.0000° W"]  
}
```

In the underlying infrastructure built in Hopsworks, metadata changes are logged as a consistent change stream to the filesystem. We further process this change stream using ePipe, a databus that both creates a consistent change stream for a distributed, hierarchical file system (HopsFS) and eventually delivers the correctly ordered stream with low latency to downstream clients [33]. One of the downstream clients is Elasticsearch. This allows us to provide full-text search capabilities with eventual consistency for all the added metadata.

As Figure 12 depicts, changes to the filesystem, including the extended metadata are logged into NDB and processed by ePipe. The ePipe service subscribes to these changes and replicates these in an eventual consistent manner to configurable endpoints. EPipe also allows for data enrichment of the change events with additional information from the database. By default, Hopsworks comes with two configured endpoints: Elasticsearch and Apache Hive [14], a scalable data warehouse built on top of Apache Hadoop.

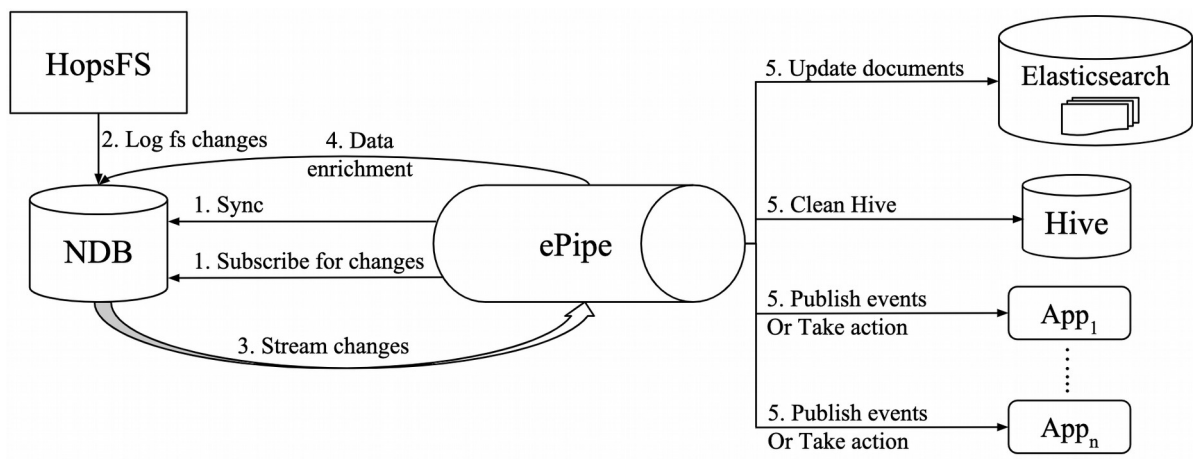


Figure 12. ePipe architecture

4.1 Search example

The demonstrator input data is stored in a dataset within Hopsworks named *eodata*. When the dataset was created, the searchable option was set. That means the dataset metadata using the mechanism described above replicated its metadata to the search engine mechanism in Hopsworks. Figures 13 and 14 below show the search results for the term *eodata* across all projects in Hopsworks.

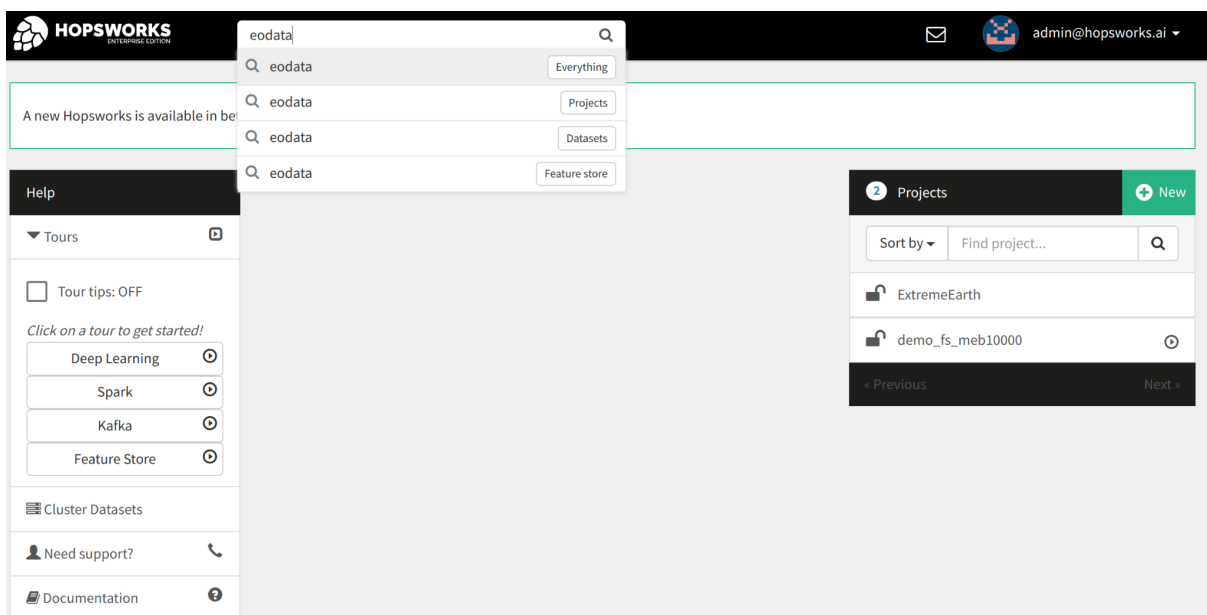


Figure 13. Searching for term eodata

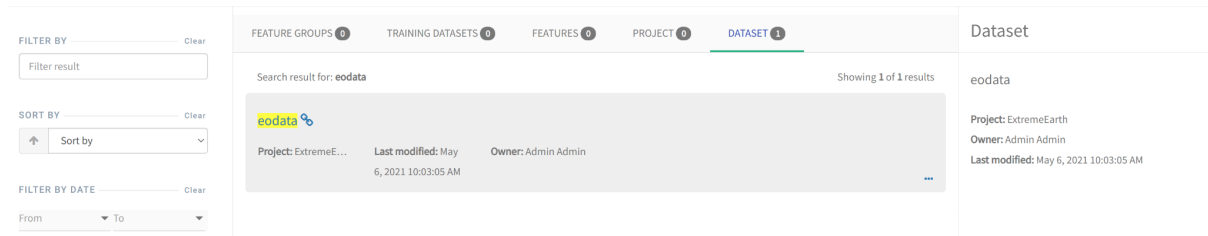


Figure 14. Search results when searching for term eodata

4.2 Tagging example

The following figures demonstrate how to create tags from the Hopworks Administration UI, how to view the details of a specific tag in the Administration, and finally how feature search results are presented to users.

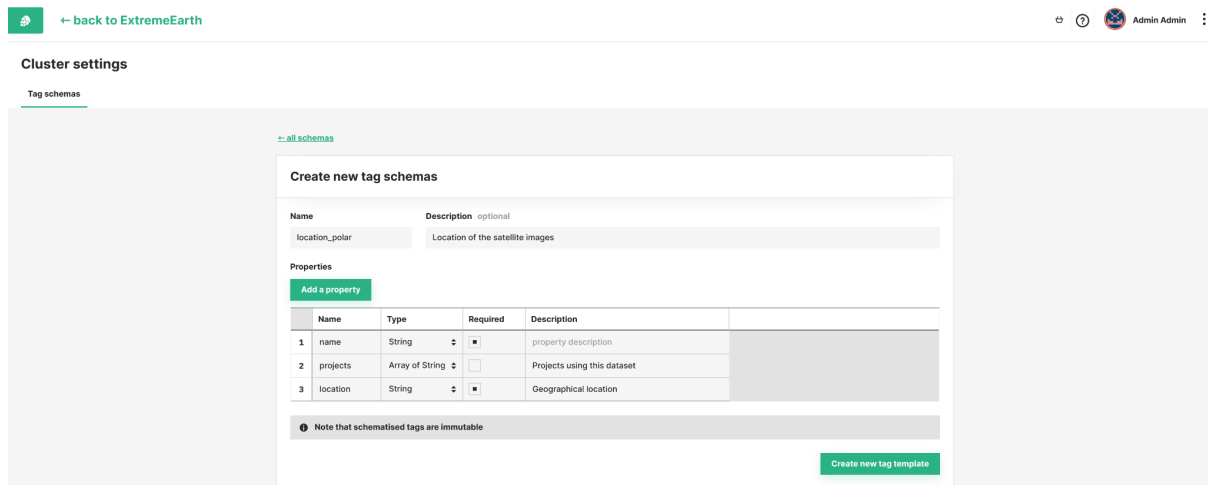


Figure 15. Admin UI to create a tag name location_polar in the new Hopworks UI



Figure 16. Viewing a tag named “location” in the Admin UI

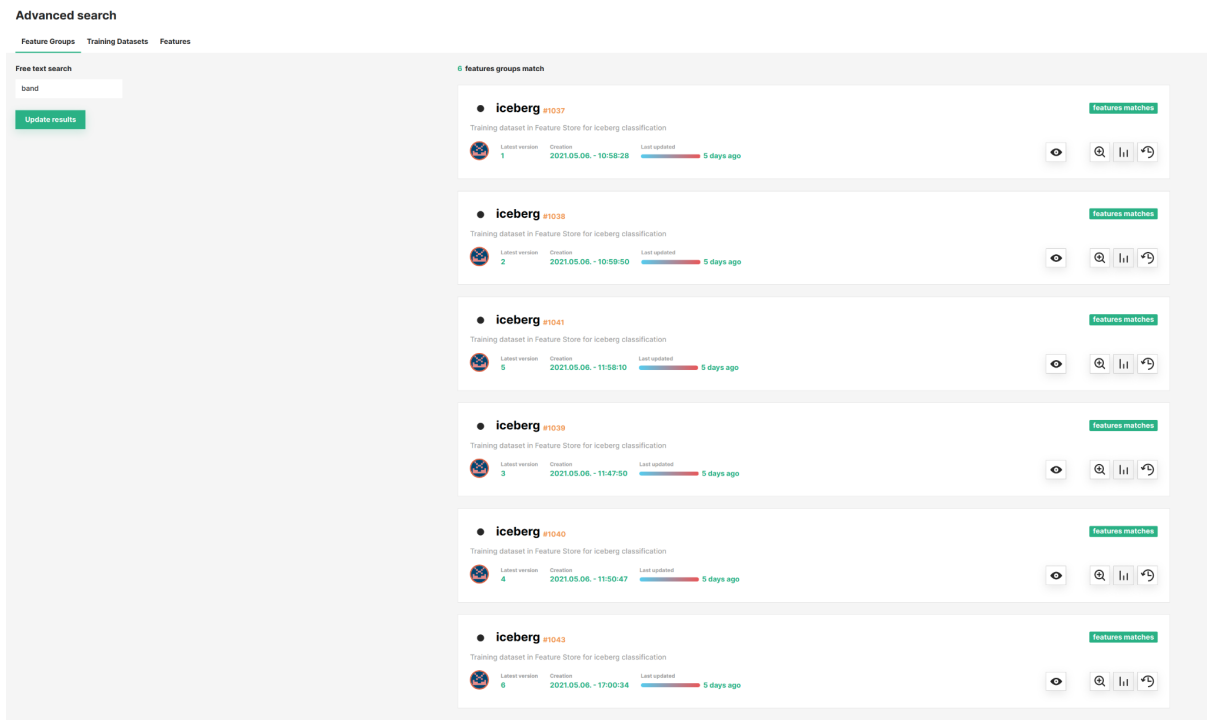


Figure 17. Search results when searching for feature “band”

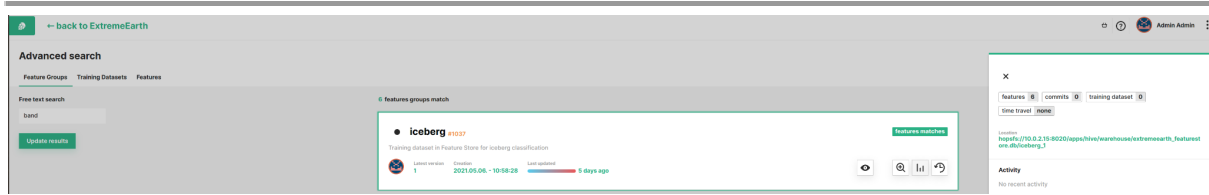


Figure 18. Search results when searching for feature “band”

5. Data Ingestion

The first step in building a scalable Deep Learning pipeline is to locate the sources where the input data reside. Then, processes need to be established that ingest, that is copy or move, data from these sources into the platform where the DL pipeline runs. These sources can be quite diverse in the format they use to store data and the protocols they implement to deliver data over to other systems. Such sources include raw data which can come from devices connected to the Internet of Things (IoT), images from satellites, structured data from data warehouses, financial transactions from real-time systems, social media, etc. Figure 19 builds on figure 1 to illustrate wherein the pipeline these external systems reside.

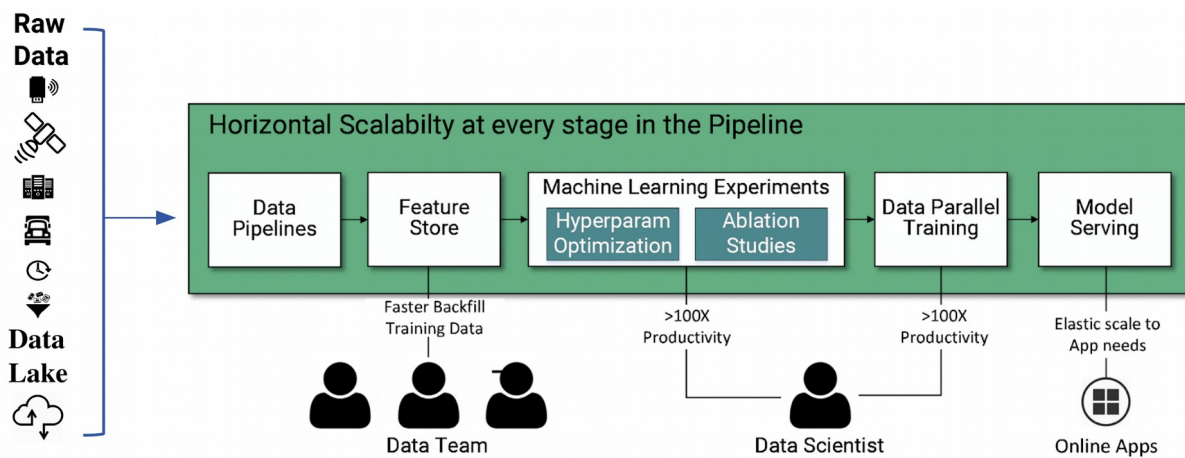


Figure 19. Data Ingestion sources for a DL pipeline

In the context of ExtremeEarth, we show the different ways in which Hopsworks has been extended to make satellite imagery data easily ingested in the platform for further processing.

5.1. Demonstrator Dataset

The scope of this deliverable is to demonstrate the extended Hopsworks platform and how it can be utilized for building deep learning pipelines with EO data. WP4 “The Food Security Use Case” and WP5 “The Polar Use Case” make use of Hopsworks with big training datasets developed for this project. As for this deliverable’s demonstrator, it was decided to use a public and free dataset related to EO data and the Polar use case in particular. The input dataset for the DL pipeline of this demonstrator is the “Statoil/C-CORE Iceberg Classifier Challenge - Ship or iceberg, can you decide from space?” [37]. It is hosted by Kaggle which is an online community of data scientists and machine learners and is distributed for free.

The schema for the Statoil dataset is presented in Figure 20. The data is in json format and contains 1604 images. For each image in the dataset, we have the following information:

- id - the id of the image.
- band_1, band_2 - the flattened image data. Each band has 75x75

-
- pixel values in the list, so the list has 5625 elements. Band 1 and Band 2 are signals characterized by radar backscatter produced from the polarizations to HH (transmit/receive horizontally) and HV (transmitted horizontally and received vertically).
 - inc_angle - the incidence angle of which the image was taken.
 - is_iceberg - set to 1 if it is an iceberg, and 0 if it is a ship.

```
1 {  
2   "band_1": {  
3     "feature": "fixed_len",  
4     "type": "float",  
5     "shape": [  
6       5625  
7     ]  
8   },  
9   "band_2": {  
10    "feature": "fixed_len",  
11    "type": "float",  
12    "shape": [  
13      5625  
14    ]  
15  },  
16  "band_avg": {  
17    "feature": "fixed_len",  
18    "type": "float",  
19    "shape": [  
20      5625  
21    ]  
22  },  
23  "id": {  
24    "feature": "var_len",  
25    "type": "string"  
26  },  
27  "inc_angle": {  
28    "feature": "var_len",  
29    "type": "string"  
30  },  
31  "is_iceberg": {  
32    "feature": "fixed_len",  
33    "type": "int"  
34  }  
35 }
```

Figure 20. Schema of the Statoil demonstrator dataset

5.2. Accessing EO Data from Hopsworks

In the context of ExtremeEarth, Hopsworks is deployed on a DIAS (CREODIAS) where EO data required for this project resides. In Creodias, EO data is made available via an object store where it can be accessed via the S3 protocol implemented by OpenStack Swift [4], or it can be accessed via standardized web services such as WMS/WMTS/WCS/WFS [3]. Regarding the latter, using a web client to access data is a well-known practice in the industry, and existing solutions can be used directly in applications developed and deployed on Hopsworks and it is beyond the scope of this deliverable to describe an example application for such a scenario.

Hopsworks has been extended to allow arbitrary Python libraries to be installed by a self-service User Interface (UI). As a result, Hopsworks users can easily install and use from within Jupyter notebooks and PySpark jobs the boto3 library [7], to access EO data from Python programs via the S3 protocol.

Figure 21 below demonstrates how to list the contents of an object store S3 bucket on CREODIAS containing Sentinel-2 data. The boto3 API also provides methods for downloading data. As a result, users developing DL pipelines in Hopsworks can easily list and download EO data for further processing.

```
In [1]: import boto3
# import the boto3 library from the project's Python environment

Starting Spark application



| ID | YARN Application ID            | Kind    | State | Spark UI             | Driver log           | Current session? |
|----|--------------------------------|---------|-------|----------------------|----------------------|------------------|
| 0  | application_1552746274750_0001 | pyspark | idle  | <a href="#">Link</a> | <a href="#">Link</a> | ✓                |



SparkSession available as 'spark'.

In [3]: # Credentials for accessing the object store.
# Since the Hops data platform is installed on the same infrastructure as the object store (DIAS),
# directly accessing the EO data is available for free.

access_key='anystring'
secret_key='anystring'
host='http://data.cloudferro.com'

s3=boto3.resource('s3',aws_access_key_id=access_key,
aws_secret_access_key=secret_key, endpoint_url=host,)

bucket=s3.Bucket('DIAS')
# The EO data bucket to explore.
# Prints the name and path to the data stored under a particular Sentinel-2 directory
for obj in bucket.objects.filter(Prefix='Sentinel-2/MSI/L1C/2016/02/03/S2A_OPER_PRD_MSIL1C_PDMC_20160204T043037_R08'):
    print('{0}:{1}'.format(bucket.name, obj.key))

DIAS:Sentinel-2/MSI/L1C/2016/02/03/S2A_OPER_PRD_MSIL1C_PDMC_20160204T043037_R085_V20160203T203520_20160203T203520
.SAFE/
DIAS:Sentinel-2/MSI/L1C/2016/02/03/S2A_OPER_PRD_MSIL1C_PDMC_20160204T043037_R085_V20160203T203520_20160203T203520
.SAFE/DATASTRIP/
DIAS:Sentinel-2/MSI/L1C/2016/02/03/S2A_OPER_PRD_MSIL1C_PDMC_20160204T043037_R085_V20160203T203520_20160203T203520
.SAFE/DATASTRIP/S2A_OPER_MSI_L1C_DS_SGS_20160203T220909_S20160203T203520_N02_01/
DIAS:Sentinel-2/MSI/L1C/2016/02/03/S2A_OPER_PRD_MSIL1C_PDMC_20160204T043037_R085_V20160203T203520_20160203T203520
.SAFE/DATASTRIP/S2A_OPER_MSI_L1C_DS_SGS_20160203T220909_S20160203T203520_N02_01/S2A_OPER_MTD_L1C_DS_SGS_2016020
3T220909_S20160203T203520.xml
DIAS:Sentinel-2/MSI/L1C/2016/02/03/S2A_OPER_PRD_MSIL1C_PDMC_20160204T043037_R085_V20160203T203520_20160203T203520
.SAFE/GRANULE/
DIAS:Sentinel-2/MSI/L1C/2016/02/03/S2A_OPER_PRD_MSIL1C_PDMC_20160204T043037_R085_V20160203T203520_20160203T203520
.SAFE/GRANULE/S2A_OPER_MSI_L1C_TL_SGS_20160203T220909_A003228_T54CWR_N02_01/
DIAS:Sentinel-2/MSI/L1C/2016/02/03/S2A_OPER_PRD_MSIL1C_PDMC_20160204T043037_R085_V20160203T203520_20160203T203520
.SAFE/GRANULE/S2A_OPER_MSI_L1C_TL_SGS_20160203T220909_A003228_T54CWR_N02_01/AUX_DATA/
DIAS:Sentinel-2/MSI/L1C/2016/02/03/S2A_OPER_PRD_MSIL1C_PDMC_20160204T043037_R085_V20160203T203520_20160203T203520
.SAFE/GRANULE/S2A_OPER_MSI_L1C_TL_SGS_20160203T220909_A003228_T54CWR_N02_01/AUX_DATA/S2A_OPER_AUX_ECMWFT_SGS_20
160203T220909_V20160203T180000_20160204T060000
DIAS:Sentinel-2/MSI/L1C/2016/02/03/S2A_OPER_PRD_MSIL1C_PDMC_20160204T043037_R085_V20160203T203520_20160203T203520
.SAFE/GRANULE/S2A_OPER_MSI_L1C_TL_SGS_20160203T220909_A003228_T54CWR_N02_01/AUX_DATA/
```

Figure 21. Accessing Sentinel-2 EO data with boto3 library and Jupyter

notebooks

Another way to directly access data via the S3 protocol is to use the Apache Spark connector to S3 [8]. Security configuration parameters such as Access Key and Secret Key can be set in the Hopsworks UI before launching a Jupyter notebook or a PySpark job.

```
In [4]: # Read image data with PySpark into a DataFrame via the s3fs protocol from the object store
df = spark.read.format("image").option("dropInvalid", "true").load("file:///eodata/Landsat-5/TM/L1T/2011/11/11/LS05")

In [8]: #count the number of images read
df.count()

1

In [9]: #print the schema of the dataframe
df.printSchema()

root
|-- image: struct (nullable = true)
|   |-- origin: string (nullable = true)
|   |-- height: integer (nullable = true)
|   |-- width: integer (nullable = true)
|   |-- nChannels: integer (nullable = true)
|   |-- mode: integer (nullable = true)
|   |-- data: binary (nullable = true)

In [10]: #show information about the read image
df.select("image.origin", "image.width", "image.height").show(truncate=False)

+-----+-----+-----+
|origin|width|height|
+-----+-----+-----+
|file:///eodata/Landsat-5/TM/L1T/2011/11/11/LS05_RKSE_TM_GTC_1P_20111111T093819_20111111T093847_147313_0191_0025_1E1E/LS05_RKSE_TM_GTC_1P_20111111T093819_20111111T093847_147313_0191_0025_1E1E.BP.PNG|1448|1448|
+-----+-----+-----+
```

Figure 22. Accessing local EO data from a PySpark program

EO data can also easily be accessed directly via the local filesystem. That is achieved by utilizing the s3fs protocol to mount the S3 compatible object store containing the EO data to the local filesystem on the operating systems Hopsworks is installed on. That makes it a lot easier for end-users to write applications that read for example Sentinel images and then proceed to perform some processing on them. The disadvantage of this approach is that accessing data through s3fs does not perform as fast as directly accessing the data via S3. However, it greatly depends on each use case whether this performance limitation affects the end users' applications. In the domain of this report, most applications are expected to be batch in nature, and directly accessing EO data from the object store is expected to be done only at the beginning of the DL pipeline. Then, EO data is preprocessed and new data is generated which is to be used in later stages of the DL pipeline. Figure 22 demonstrates how users can read a Sentinel-2 image stored in an S3 bucket directly into a PySpark application, in this case, a PySpark dataframe.

Users can also have EO data stored directly in HopsFS, the distributed file system Hopsworks is built on. A reason to do so might include wanting to share data, particularly data that has

been generated by applications that run on Hopsworks, across projects. D1.1 provides an in-depth guide of how data can be shared among projects. Another reason might be that EO data needed for a particular pipeline is available online or on the DIAS. Therefore such data needs to be uploaded onto Hopsworks. The latter provides a REST API endpoint under “POST /project/{projectId}/dataset/upload/{path}” [6] to upload any file types and have them available from within a Hopsworks project. Figure 23 shows how to upload a file into a project named ExtremeEarthTEP and into a dataset named EOData. Source code is available at [9].

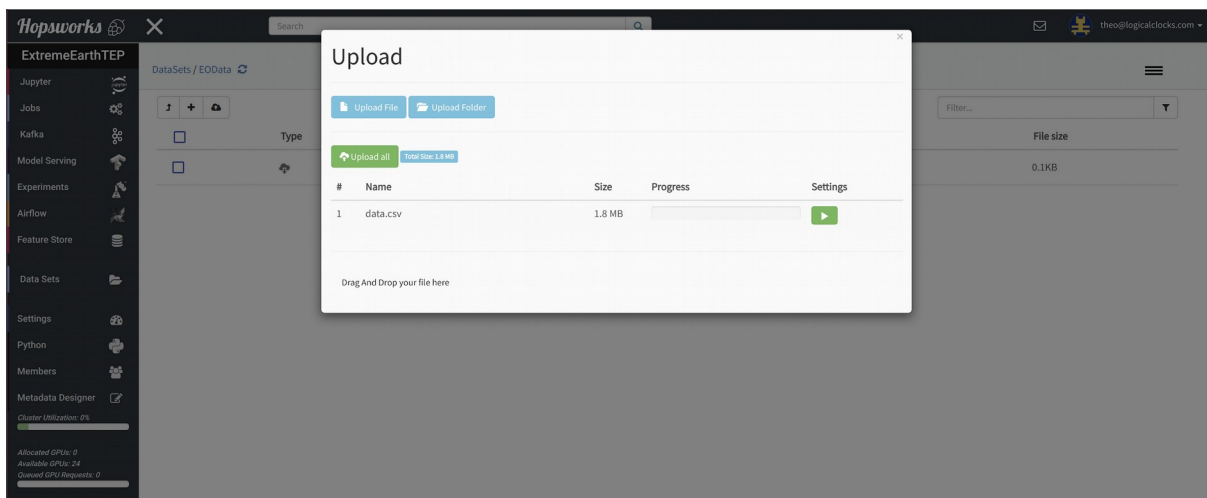


Figure 23. Uploading a file into an EOData dataset in the ExtremeEarthTEP project

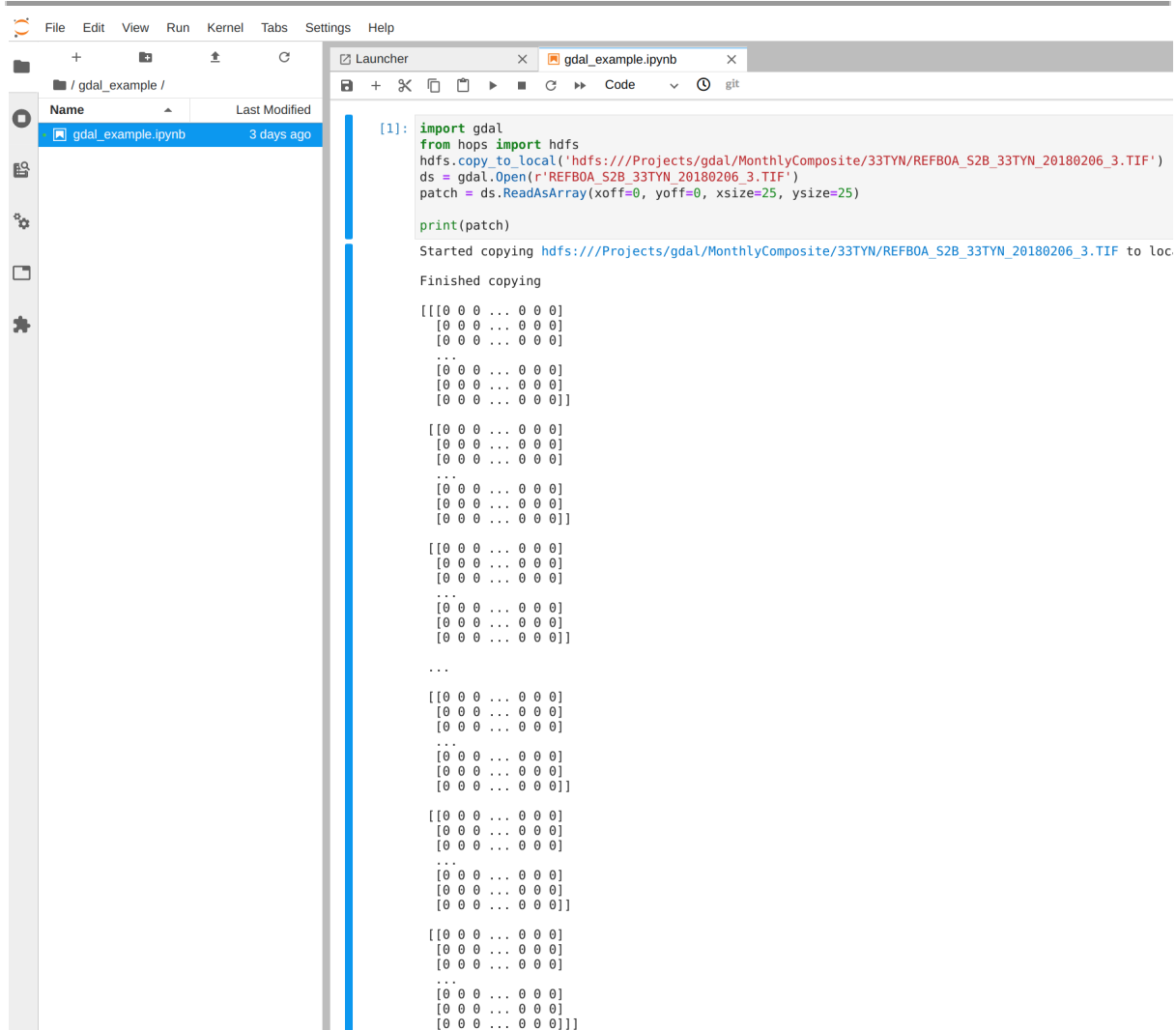
In the demo DL pipeline, the input EO dataset is stored in a Hopsworks dataset “Statoil” of a project “ExtremeEarth”.

6. EO data pre-processing

6.1 EO data pre-processing with Python

Oftentimes satellite data needs to be processed before being provided as input to machine learning algorithms. By utilizing the Python support in Hopsworks as described in Section 3 Development Environment, data scientists in ExtremeEarth can use programs such as GDAL to process EO data. GDAL is a translator library for raster and vector geospatial data formats [59] and it comes with a Python package and extensions are a number of tools for programming and manipulating the GDAL Geospatial Data Abstraction Library [60].

In the example below, some EO data in the form of a .TIF file is available in a Hopsworks dataset. The data scientist can then install gdal from the project’s Python environment and use it directly either from a Jupyter notebook or a Hopsworks Job. Figure 24 demonstrates the first option, where a Jupyter notebook is used to read the .TIF file from a dataset, open it using GDAL, read it as an array, and print a patch of the image.



```

[1]: import gdal
from hops import hdf5
hdfs.copy_to_local('hdfs:///Projects/gdal/MonthlyComposite/33TYN/REFB0A_S2B_33TYN_20180206_3.TIF')
ds = gdal.Open(r'REFB0A_S2B_33TYN_20180206_3.TIF')
patch = ds.ReadAsArray(xoff=0, yoff=0, xsize=25, ysize=25)

print(patch)

Started copying hdfs:///Projects/gdal/MonthlyComposite/33TYN/REFB0A_S2B_33TYN_20180206_3.TIF to loc
Finished copying

[[[0 0 0 ... 0 0 0]
  [0 0 0 ... 0 0 0]
  [0 0 0 ... 0 0 0]
  ...
  [0 0 0 ... 0 0 0]
  [0 0 0 ... 0 0 0]
  [0 0 0 ... 0 0 0]]

[[[0 0 0 ... 0 0 0]
  [0 0 0 ... 0 0 0]
  [0 0 0 ... 0 0 0]
  ...
  [0 0 0 ... 0 0 0]
  [0 0 0 ... 0 0 0]
  [0 0 0 ... 0 0 0]]

[[[0 0 0 ... 0 0 0]
  [0 0 0 ... 0 0 0]
  [0 0 0 ... 0 0 0]
  ...
  [0 0 0 ... 0 0 0]
  [0 0 0 ... 0 0 0]
  [0 0 0 ... 0 0 0]]

[[[0 0 0 ... 0 0 0]
  [0 0 0 ... 0 0 0]
  [0 0 0 ... 0 0 0]
  ...
  [0 0 0 ... 0 0 0]
  [0 0 0 ... 0 0 0]
  [0 0 0 ... 0 0 0]]

[[[0 0 0 ... 0 0 0]
  [0 0 0 ... 0 0 0]
  [0 0 0 ... 0 0 0]
  ...
  [0 0 0 ... 0 0 0]
  [0 0 0 ... 0 0 0]
  [0 0 0 ... 0 0 0]]

[[[0 0 0 ... 0 0 0]
  [0 0 0 ... 0 0 0]
  [0 0 0 ... 0 0 0]
  ...
  [0 0 0 ... 0 0 0]
  [0 0 0 ... 0 0 0]
  [0 0 0 ... 0 0 0]]

[[[0 0 0 ... 0 0 0]
  [0 0 0 ... 0 0 0]
  [0 0 0 ... 0 0 0]
  ...
  [0 0 0 ... 0 0 0]
  [0 0 0 ... 0 0 0]
  [0 0 0 ... 0 0 0]]

```

Figure 24. Jupyter notebook processing monthly composists with GDAL in Python

6.2 EO data pre-processing with Docker and Kubernetes

Hopsworks has been extended in ExtremeEarth to provide support for working with arbitrary programming languages and frameworks when processing EO data by enabling users to run arbitrary Docker containers on Kubernetes via the Hopsworks Jobs service. The motivation behind this functionality is that users might need to use tools and frameworks that are not necessarily available in the Python anaconda environment of the project, such as Java or C++ tools related to Remote Sensing and EO data.

As of the time of writing, Hopsworks has integrated Docker version 19.03.8 and Kubernetes version 1.18.8. When users submit a Docker job in Hopsworks, the latter securely connects to the Kubernetes cluster and submits a Kubernetes job that contains metadata and security material (TLS certificates) unique for this job. Hopsworks is then able to monitor the job and collect logs back to Hopsworks datasets. The entire Docker and Kubernetes infrastructure stack is transparent to users, as they only need to interact with the Hopsworks UI and the

client APIs. Figure 25 displays the software stack integrated into Hopsworks that enables users to run jobs and notebooks, including the Docker job type used for the EO data pre-processing described in this section.

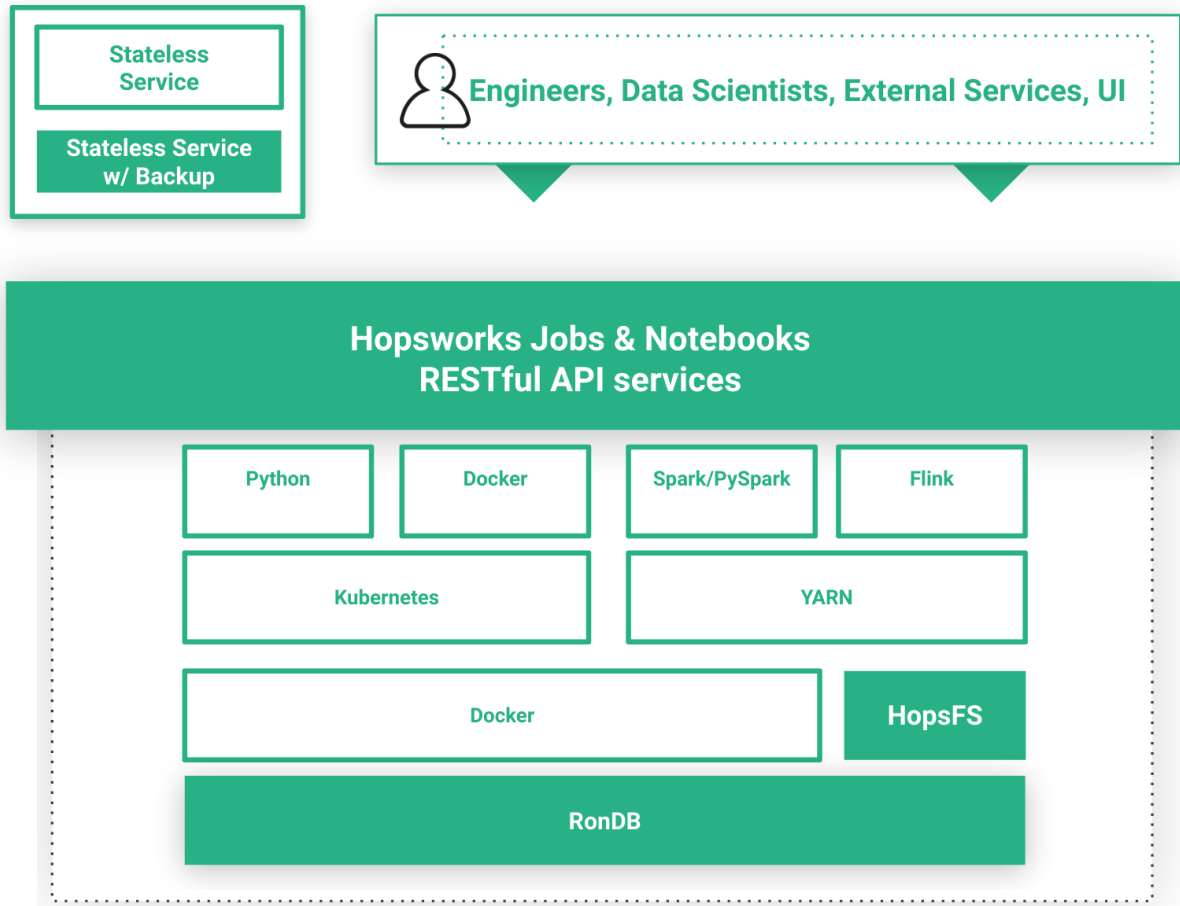


Figure 25. Hopsworks Jobs and notebooks services infrastructure

Users can specify the following properties for a Docker job:

- *Docker image*: The location of the Docker image. Currently only publicly accessible docker registries are supported.
- *Docker command*: The command to run the Docker container with.
- *Docker command arguments*: Comma-separated list of input arguments of the Docker command.
- *Output path*: The location in Hopsworks datasets where the output of the Job will be persisted, if the programs running inside the container redirect their output to the same container-local path. For example, if the output path is set to `/Projects/myproject/Resources` and the container runs the command `echo "hello" >> /Projects/myproject/Resources/hello.txt`, then the Hopsworks job upon job

completion will copy the entire content of the /Projects/myproject/Resources from the docker container to the corresponding path with the same name under Datasets.

- *Environment variables*: Comma-separated list of environment variables to be set for the Docker container.
- *Volumes*: Comma-separated list of volumes to be mounted with the Docker job.
- *User id / Group Id*: Provide the uid and gid to run the Docker container with. For further details, look into the *Admin options* below.

Certain options are available to Hopsworks users with the role HOPS_ADMIN only, as these are applied cluster-wide:

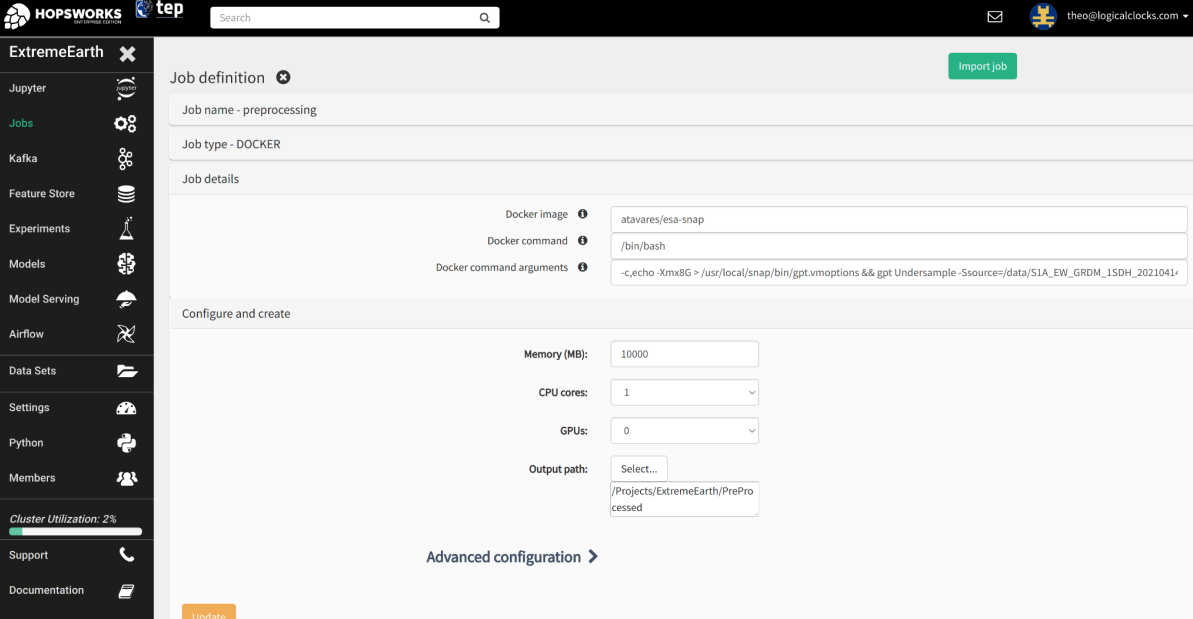
- *docker_job_mounts_list*: Comma-separated list of host paths jobs are allowed to mount. Default is empty.
- *docker_job_mounts_allowed*: Whether mounting volumes are allowed. Allowed values: true/false. Default is false.
- *docker_job_uid_strict*: Enable or disable strict mode for uid/gid of docker jobs. In strict mode, users cannot set the uid/gid of the job. The default is true. If false and users do not set uid and gid, the container will run with the uid/gid set in the Dockerfile.

An example of a platform with a variety of tools for EO data processing in ESA's SNAP toolbox [56]. Docker images are available on Dockerhub with the ESA SNAP toolbox and GPT and for this demonstrator the *atavares/esa-snap* was selected [58]. As Hopsworks is running within the TEP and Creodias infrastructure, it has been extended to access the plethora of EO data provided by these services, as described in section 5 Data Ingestion. In this example, a Docker job is setup in Hopsworks that uses the GPT tool from the *atavares/esa-snap* Docker image that is pulled from Dockerhub, reads Sentinel-1 data from the Creodias provided data storage, runs the gpt tool to undersample the input data and outputs the data in the Hopsworks datasets browser. Figure 26 shows the input data that is mounted as a volume with the Docker job so that the GPT tool can read and process it.

```
[eouser@hopsworks-2 ~]$ ls -l /eodata/Sentinel-1/SAR/GRD/2021/04/14/S1A_EW_GRDM_1SDH_20210414T064754_20210414T064858_037443_0469F8_FFAD.SAFE
total 39
dr-xr-xr-x. 1 root root    0 Apr 14 10:27 annotation
-r-xr-xr-x. 1 root root 21640 Apr 14 10:27 manifest.safe
dr-xr-xr-x. 1 root root    0 Apr 14 10:27 measurement
dr-xr-xr-x. 1 root root    0 Apr 14 10:27 preview
-r-xr-xr-x. 1 root root 15109 Apr 14 10:27 S1A_EW_GRDM_1SDH_20210414T064754_20210414T064858_037443_0469F8_FFAD.SAFE-report-20210414T100747.pdf
dr-xr-xr-x. 1 root root    0 Apr 14 10:27 support
```

Figure 26. EO data from CREODIAS in the Hopsworks PolarTEP cluster

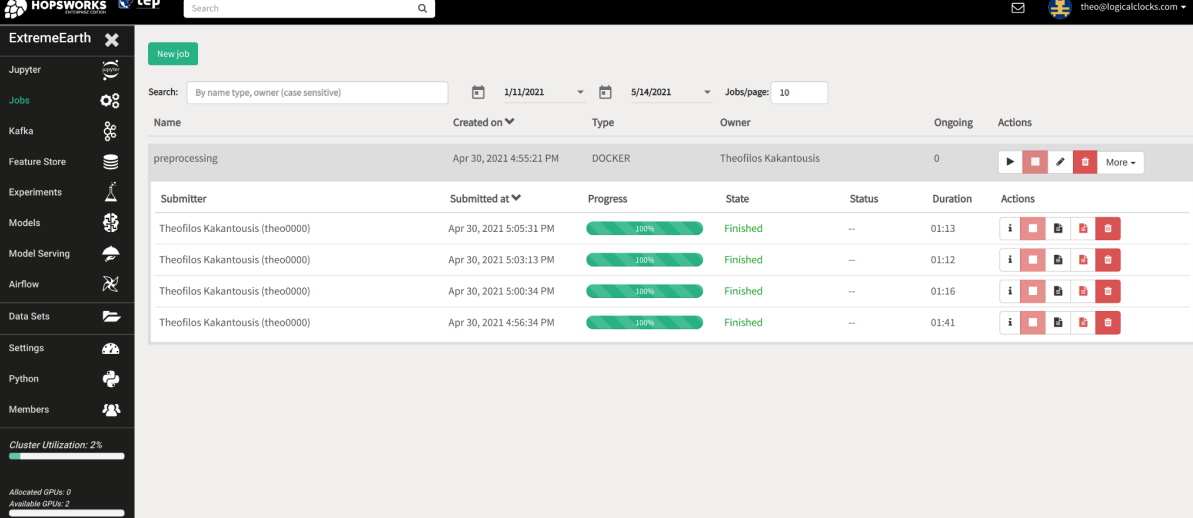
Figures 27-29 show the configuration of this particular job, an overview of previous executions of the job, and the output data in the datasets browser.



The screenshot shows the 'Job definition' page in the HOPSWORKS ExtremeEarth interface. The left sidebar contains navigation links for Jupyter, Jobs, Kafka, Feature Store, Experiments, Models, Model Serving, Airflow, Data Sets, Settings, Python, and Members. The main content area is titled 'Job definition' and includes a search bar, a user profile icon, and a green 'Import job' button. The configuration details are as follows:

- Job name:** preprocessing
- Job type:** DOCKER
- Job details:**
 - Docker image:** atavares/esa-snap
 - Docker command:** /bin/bash
 - Docker command arguments:** -c echo -Xmx8G > /usr/local/snap/bin/gpt.vmoptions && gpt Undersample -Source=/data/S1A_EW_GRDM_1SDH_2021041
- Configure and create:**
 - Memory (MB):** 10000
 - CPU cores:** 1
 - GPUs:** 0
 - Output path:** Select... (dropdown menu showing /Projects/ExtremeEarth/PreProcessed)
- Advanced configuration >**
- Update** button

Figure 27. Docker job configuration for EO data pre-processing with ESA/SNAP and GPT



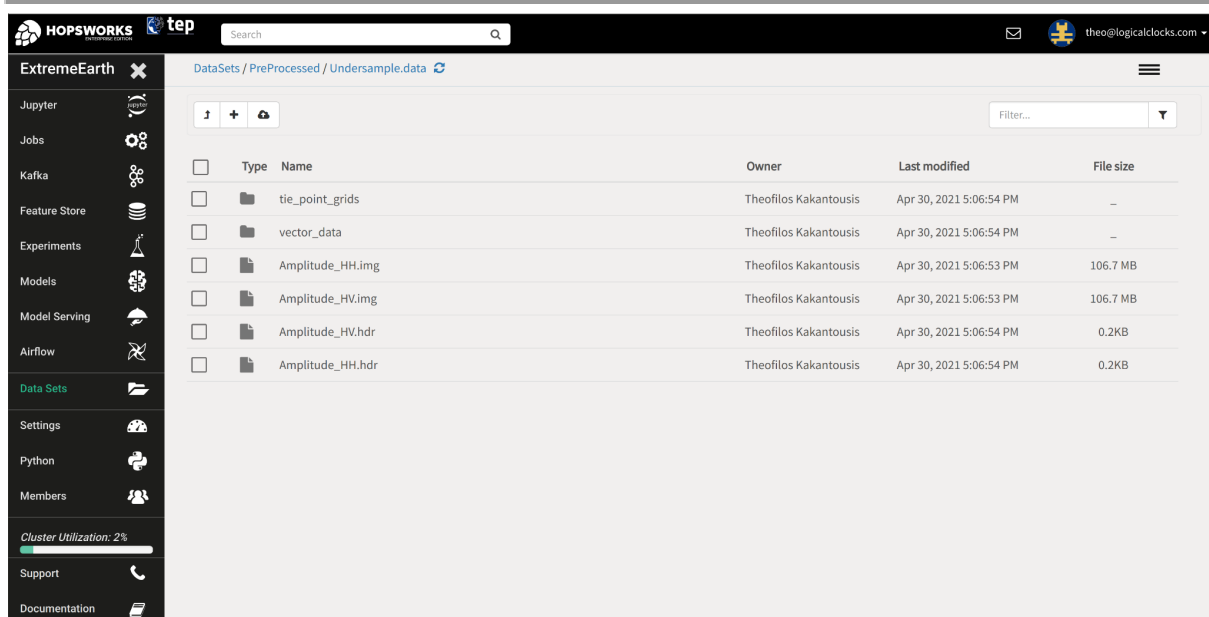
The screenshot shows the 'Jobs' overview page in the HOPSWORKS ExtremeEarth interface. The left sidebar is the same as in Figure 27. The main content area has a 'New job' button and a search bar. Below the search bar is a table of job executions. The table has columns for Name, Created on, Type, Owner, Ongoing, and Actions. The data is as follows:

Name	Created on	Type	Owner	Ongoing	Actions
preprocessing	Apr 30, 2021 4:55:21 PM	DOCKER	Theofilos Kakantousis	0	[Play] [Stop] [Refresh] [More]

Below the main table is a detailed view of the job executions with columns: Submitter, Submitted at, Progress, State, Status, Duration, and Actions. The data is as follows:

Submitter	Submitted at	Progress	State	Status	Duration	Actions
Theofilos Kakantousis (theo0000)	Apr 30, 2021 5:05:31 PM	100%	Finished	--	01:13	[Info] [Stop] [Refresh] [More]
Theofilos Kakantousis (theo0000)	Apr 30, 2021 5:03:13 PM	100%	Finished	--	01:12	[Info] [Stop] [Refresh] [More]
Theofilos Kakantousis (theo0000)	Apr 30, 2021 5:00:34 PM	100%	Finished	--	01:16	[Info] [Stop] [Refresh] [More]
Theofilos Kakantousis (theo0000)	Apr 30, 2021 4:56:34 PM	100%	Finished	--	01:41	[Info] [Stop] [Refresh] [More]

Figure 28. Docker job executions overview



Type	Name	Owner	Last modified	File size
Folder	tie_point_grids	Theofilos Kakantousis	Apr 30, 2021 5:06:54 PM	–
Folder	vector_data	Theofilos Kakantousis	Apr 30, 2021 5:06:54 PM	–
Image	Amplitude_HH.img	Theofilos Kakantousis	Apr 30, 2021 5:06:53 PM	106.7 MB
Image	Amplitude_HV.img	Theofilos Kakantousis	Apr 30, 2021 5:06:53 PM	106.7 MB
Header	Amplitude_HV.hdr	Theofilos Kakantousis	Apr 30, 2021 5:06:54 PM	0.2KB
Header	Amplitude_HH.hdr	Theofilos Kakantousis	Apr 30, 2021 5:06:54 PM	0.2KB

Figure 29. EO data from Creodias in the Hopsworlds PolarTEP cluster

7. Feature Engineering and Data Validation

7.1 Feature Store

Feature engineering can be described as the process with which domain knowledge on ingested data is applied, in order to create features that are used in further stages of the DL pipeline (Training). With the continuous growth in input data and increased complexity of DL pipelines, arose the need for a framework that facilitates features engineering and reduces the complexity of managing features.

To improve the management of curated feature data, Hopsworlds has been extended with a new framework named Feature Store. The Feature Store acts as the central management layer for curated data in an organization and it serves as the interface between data engineering and data science teams. The motivation of feature engineering is to generate reusable features that can be shared across different teams in an organization and can facilitate developing new ML models, as depicted in Figure 30. Benefits the Feature Store brings include reuse of features across pipelines, feature discoverability with free-text search across an organization's feature data, applying software engineering principles onto machine learning features with versioning, documentation, and access-control, time-travel by fetching past feature data that were used for training particular model, scalability in terms of being able to manage multiple terabytes or even bigger feature datasets, analysis so data scientists can gather useful insights regarding data distribution, correlation, etc. [13].

Raw/Structured Data

Models

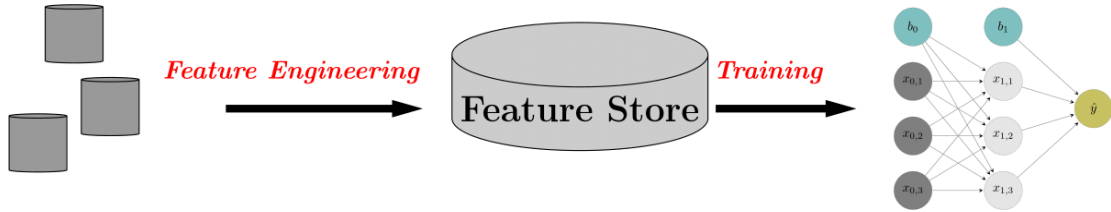


Figure 30. The feature store as the link between Feature Engineering and Training

To achieve all the aforementioned properties, the Feature Store is implemented on scalable, fault-tolerant services. Offline data is stored in Apache Hive [14], a scalable data warehouse built on top of Apache Hadoop, and online data is stored in MySQL Cluster. Offline features can be used for training/experimentation and are used mostly in batch-oriented use cases where past feature data can be fetched or huge volumes of feature data can be analysed to generate statistics. Online features need to be accessible in real-time for pipelines that need to get data at prediction time. Besides storing data, the Feature Store utilizes the Spark integration in Hopsworks, described in the previous sections, to compute and analyze features. Figure 31 shows the main components of the Hopsworks Feature Store.

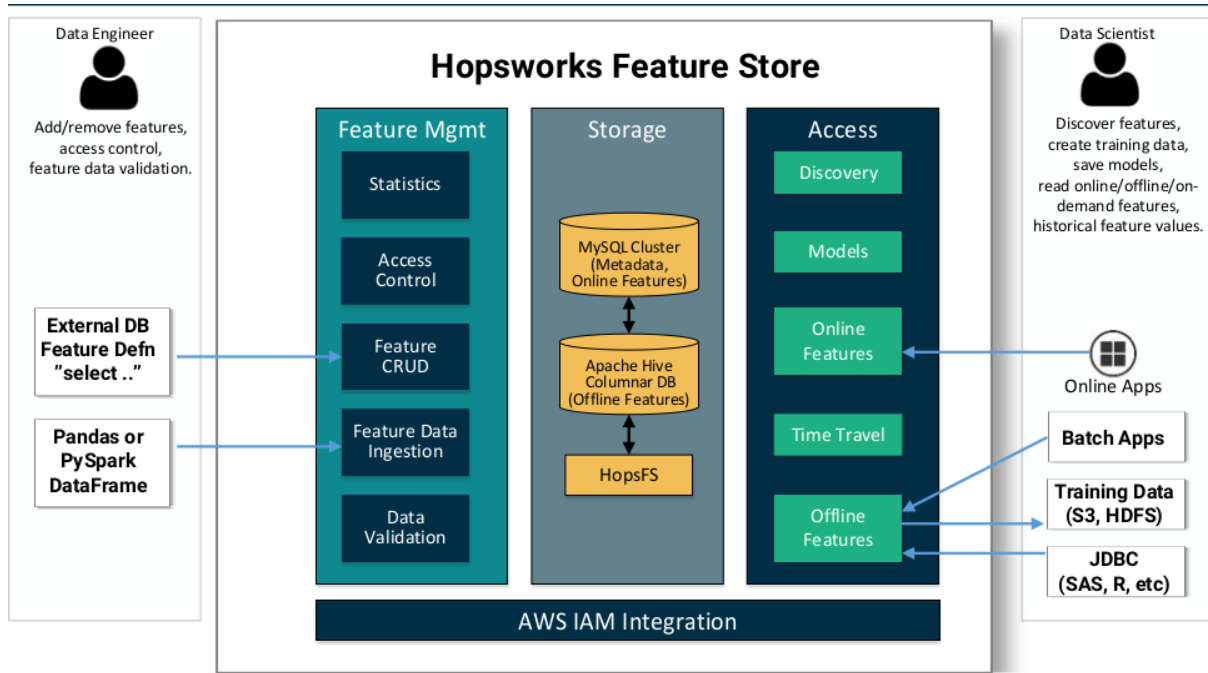


Figure 31. Hopsworks Feature Store Architecture[13]

In the previous section, the Statoil/C-CORE Iceberg Classifier Challenge dataset used in this demo was made available in Hopsworks by uploading it via the Hopsworks datasets browser UI. The second step in the pipeline is to do some feature engineering of the input dataset,

which is demonstrated in Figure 32. Initially, the raw dataset is read from its location in HopsFS “hdfs://127.0.0.1:8020/Projects/ExtremeEarth/eodata/train.json” into a Pandas dataframe, an open-source Python library for data manipulation and analysis [46], in the Python program. The dataset is in json format and Pandas natively supports reading from this file format. We extended Pandas with wrapper functions to read such file formats directly from HopsFS [17]. Then a new feature band_avg is computed which is the average of the two image bands, band_1 and band_2. Finally, the new raw_train_df is saved back to a Hopsworks dataset in HopsFS. That way, the dataset is made available to further processing steps. Figure 32 demonstrates the process described so far, with the full version of the iPython notebook being available at [18].

Read the raw data

```
[3]: # read the raw data to pandas dataframe
raw_train_df = pd.read_json(train_ds_path)
```

```
[4]: raw_train_df
```

	id	...	is_iceberg
0	dfd5f913	...	0
1	e25388fd	...	0
2	58b2aaa0	...	1
3	4cfc3a18	...	0
4	271f93f4	...	0
...
1599	04e11240	...	0
1600	c7d6f6f8	...	0
1601	bbala0f1	...	0
1602	7f66bb44	...	0
1603	9d8f326c	...	0

[1604 rows x 5 columns]

Create new feature band_avg

```
[5]: # a function for taking list average
def list_avg(row):
    return [sum(x)/2 for x in zip(row['band_1'], row['band_2'])]

raw_train_df['band_avg'] = raw_train_df.apply(lambda row: list_avg(row), axis=1)
```

```
[6]: raw_train_df
```

	id	...	band_avg
0	dfd5f913	...	[-27.516239499999998, -28.346024, -29.84960749...]
1	e25388fd	...	[-21.874347999999998, -21.4524295, -20.7830205...]
2	58b2aaa0	...	[-24.737316, -24.348173, -22.762496, -21.28190...]
3	4cfc3a18	...	[-25.172013999999997, -25.301306500000003, -25...]
4	271f93f4	...	[-26.6069355, -26.712035999999998, -26.7120359...]
...
1599	04e11240	...	[-29.4237985, -29.105365, -26.472991999999998,...]
1600	c7d6f6f8	...	[-27.437631500000002, -27.400965, -27.76694599...]
1601	bbala0f1	...	[-21.723625, -23.7647725, -23.9906165, -22.930...]
1602	7f66bb44	...	[-24.262994499999998, -23.944199, -24.2661145,...]
1603	9d8f326c	...	[-22.1770305, -22.817203499999998, -23.9654685...]

[1604 rows x 6 columns]

```
[7]: #save raw train df in dataset
raw_train_df.to_json(path_or_buf='train_preprocessed_all.json', orient='records')
hdfs.coop to hdfs("train_preprocessed_all.json". DATA FOLDER . overwrite=True)
```

Figure 32. Processing the raw dataset

The next step of this stage is to read the dataset that was processed previously, and create a Feature Group in the Feature Store. Figure 33 demonstrates how this is achieved by using the Hopsworks Feature Store Python API of the hsfs Python library. In this demo, the dataset is read first from HopsFS into a PySpark dataframe *train_preprocessed_all_df* which is then inserted into the Feature Store and more specifically into a Feature Group called iceberg. A feature group is a documented and versioned group of features. In this example, automatically generating statistics has been disabled (set to False) to speed up the creating process. Statistics can be updated at any time after the Feature Group is created, either through the UI or the API.

Create and save features to the Feature Store

```
[6]: conn = hsfs.connection()
    fs = conn.get_feature_store()

    Connected. Call `.close()` to terminate connection gracefully.

[7]: icebergs_fg = fs.create_feature_group(
    "iceberg",
    time_travel_format=None,
    statistics_config=hsfs.statistics_config.StatisticsConfig(enabled=False, correlations=False, histograms=False, columns=[]),
    description="Training dataset in Feature Store for iceberg classification"
)

[8]: icebergs_fg.save(train_preprocessed_all_df)
```

Figure 33. Create and populate the Feature Group

The next step is to export the feature data into a training dataset in tfrecord format, which can be used by the next pipeline stage, Training. Figure 34 demonstrates how this is achieved by using the Python API in the same notebook.

```
[14]: # create a training dataset of TFRecord
    icebergs_train_td = fs.create_training_dataset(
        "train_tfrecords_iceberg_classification_dataset",
        statistics_config=hsfs.statistics_config.StatisticsConfig(enabled=False, correlations=False, histograms=False, columns=[]),
        data_format = "tfrecords"
    ).save(icebergs_train_df)

    VersionWarning: No version provided for creating training dataset `train_tfrecords_iceberg_classification_dataset`, incremented version to `1`.

[15]: # create a training dataset of TFRecord
    icebergs_test_td = fs.create_training_dataset(
        "test_tfrecords_iceberg_classification_dataset",
        statistics_config=hsfs.statistics_config.StatisticsConfig(enabled=False, correlations=False, histograms=False, columns=[]),
        data_format = "tfrecords"
    ).save(icebergs_test_df)
```

Figure 34. Creating training and test datasets in tfrecord format

Users are also able to interact with the Feature Store from the Hopsworks UI. Figure 35 shows the feature group overview in the Hopsworks UI.

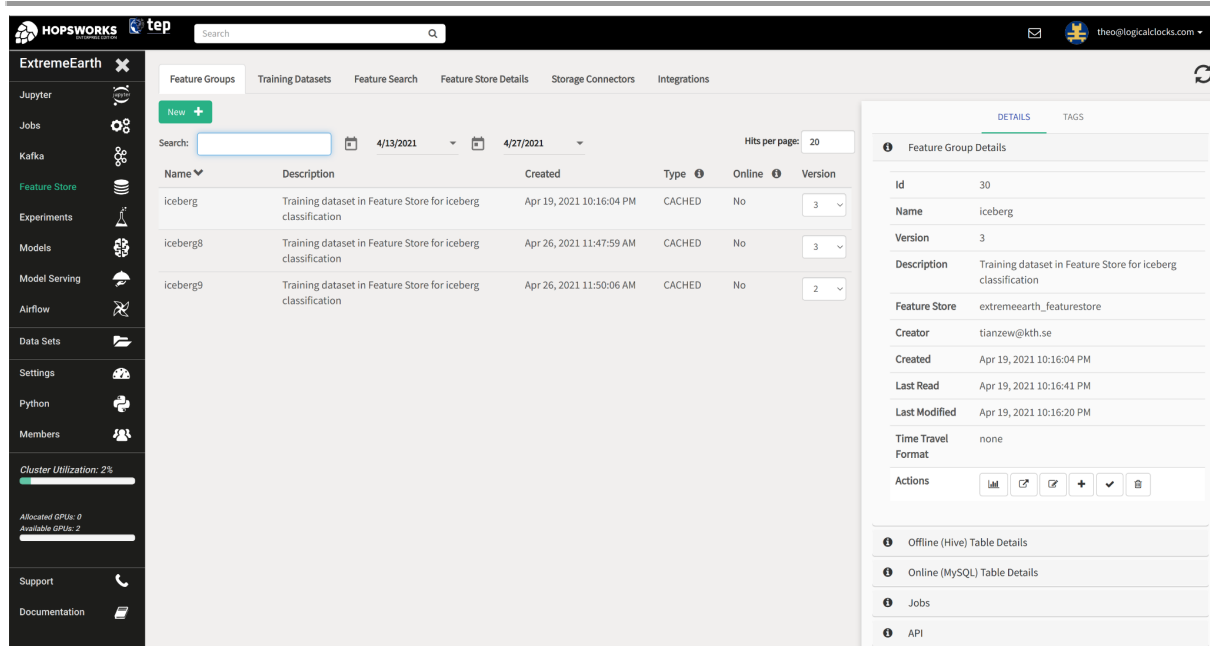


Figure 35. iceberg feature group in Hopworks

7.2 Feature Validation

Feature validation is the process of inspecting and cleansing data to be used as features in machine learning models in order to ensure their quality is sufficiently good for them to be processed by the subsequent stages of the DL pipeline. The process of performing feature validation can greatly vary in terms of implementation from among DL pipelines or among data engineers and scientists. A reason for this is that data validation is not a strict set of rules that need to be applied to ingested data, rather is a set of best practices and some common rules, derived typically from the domain of statistics.

In ExtremeEarth, there is one more constraint that needs to be taken into account when establishing a feature validation process, which is that the latter needs to be applied to large volumes of data in a distributed storage and processing environment. In addition, data validation in this DL pipeline context is applied to the feature data that reside in the Feature Store and then are extracted in the form of training or test datasets to be served as input in the Training stage. To achieve data validation at scale, the Hopworks Feature Store has been extended to support feature validation by introducing the concepts of Feature Expectations and Validation rules/results. This validation framework is built on Apache Spark and Deequ [25]. Deequ is an open-source library built on top of Apache Spark that helps developers establish data validation rules and extract useful information regarding large datasets.

To demonstrate feature validation, the validation rules applied on the ingested dataset are shown in the table below. A full list of validation rules supported in Hopworks is available at the feature validation guide of the feature store documentation page [55].

HAS_DATATYPE	ACCEPTED_ TYPE	String	-	Assert on the fraction of rows that conform to the given data type.
HAS_MAX	VALUE	Fractional	Quantitative	Assert on the max of a feature.
HAS_MIN	VALUE	Fractional	Quantitative	Assert on the min of a feature.

These rules were grouped in two feature store expectations which were then attached to the iceberg feature group as shown in the code snippet in figure 36. In particular:

1. **HAS_DATATYPE**: Asserts that the feature *id* of the iceberg feature group does not contain null values. This is asserted by setting the max allowed null values to zero. Additionally, the *is_iceberg* label is also expected to only contain numbers by setting the threshold for required numeric values of *is_iceberg* to 1.
2. **HAS_MAX**: Assertion on the maximum allowed value of the *is_iceberg* label, which is set to 1
3. **HAS_MIN**: Assertion on the minimum allowed value of the *is_iceberg* label, which is set to 0.

Create feature expectations with validation rules

```

expectation_id = fs.create_expectation("icebergs_id",
                                      description="validate inc_angle feature values",
                                      features=["id"],
                                      rules=[Rule(name="HAS_DATATYPE", level="ERROR", accepted_type="Null", max=0)])
expectation_id.save()

expectation_label = fs.create_expectation("is_iceberg",
                                          features=["is_iceberg"],
                                          description="validate is_iceberg label values",
                                          rules=[Rule(name="HAS_DATATYPE", level="ERROR", accepted_type="Integral", min=1), Rule(name="HAS_MAX", level="ERROR", accepted_type="Null", max=0)])
expectation_label.save()

expectation.rules[0].to_dict()
{'name': 'HAS_DATATYPE', 'level': 'ERROR', 'min': None, 'max': 0, 'value': None, 'pattern': None, 'acceptedType': 'Null', 'legalValues': None}
ExpectationsApi.expectation.to_dict()
{'name': 'icebergs_id', 'description': 'validate inc_angle feature values', 'features': ['id'], 'rules': [{'name': 'HAS_DATATYPE', 'level': 'ERROR', 'min': None, 'max': 0, 'value': None, 'pattern': None, 'acceptedType': 'Null', 'legalValues': None}]}
ExpectationsApi.expectation.rules[0].to_dict()
{'name': 'HAS_DATATYPE', 'level': 'ERROR', 'min': None, 'max': 0, 'value': None, 'pattern': None, 'acceptedType': 'Null', 'legalValues': None}
ExpectationsApi.expectation.payload()
{"name": "icebergs_id", "description": "validate inc_angle feature values", "features": ["id"], "rules": [{"name": "HAS_DATATYPE", "level": "ERROR", "min": null, "max": 0, "value": null, "pattern": null, "acceptedType": "Null", "legalValues": null}]}
ExpectationsApi.expectation.rules[0].to_dict()
{'name': 'HAS_DATATYPE', 'level': 'ERROR', 'min': 1, 'max': None, 'value': None, 'pattern': None, 'acceptedType': 'Integral', 'legalValues': None}
ExpectationsApi.expectation.to_dict()
{'name': 'is_iceberg', 'description': 'validate is_iceberg label values', 'features': ['is_iceberg'], 'rules': [{'name': 'HAS_DATATYPE', 'level': 'ERROR', 'min': 1, 'max': None, 'value': None, 'pattern': None, 'acceptedType': 'Integral', 'legalValues': None}]}
ExpectationsApi.expectation.payload()
{"name": "is_iceberg", "description": "validate is_iceberg label values", "features": ["is_iceberg"], "rules": [{"name": "HAS_DATATYPE", "level": "ERROR", "min": 1, "max": null, "value": null, "pattern": null, "acceptedType": "Integral", "legalValues": null}, {"name": "HAS_MAX", "level": "ERROR", "min": 1, "max": 1, "value": null, "pattern": null, "acceptedType": null, "legalValues": null}]}

```

Create and save features to the Feature Store

```

icebergs_fg = fs.create_feature_group(
    "iceberg",
    time_travel_format=None,
    statistics_config=hsfs.statistics_config.StatisticsConfig(enabled=False, correlations=False, histograms=False, columns=[]),
    expectations=[expectation_id, expectation_label],
    validation_type="STRICT",
    description="Training dataset in Feature Store for iceberg classification"
)

icebergs_fg.save(train_preprocessed_all_df)

<hsfs.feature_group.FeatureGroup object at 0x7f219dc02e90>
VersionWarning: No version provided for creating feature group `iceberg`, incremented version to `6`.

```

Figure 36. Feature expectations Python API example







Expectations ⓘ see data validation activity						
2 expectations- configured in strict mode (data is ingested only on data validation success)					<button>Attach an expectation</button>	
name	last validation			rules	features concerned	
icebergs_id	success 0	warning 0	alert 0	Datatype	id	  
is_iceberg	success 0	warning 0	alert 0	Datatype, Maximum, Minimum	is_iceberg	  

Figure 37. Feature expectation in the Hopsworks UI

The validation type is set to *Strict*, which means that if any expectation is not met then the data will be inserted into the feature group. The rest of the validation types are:

- **WARNING:** Data validation is performed and feature group is updated only if validation status is "Warning" or lower
- **ALL:** Data validation is performed and feature group is updated only if validation status is "Failure" or lower
- **NONE:** Data validation not performed on feature group

The validation results of expectations for the iceberg feature group are displayed in JSON format in figure 38 and can be accessed via the feature store API or via the Hopsworks UI.

Retreiving validation results

```
[16]: import json
[print(json.dumps(validation.to_dict(), indent=2)) for validation in icebergs_fg.get_validations()]

{
  "validationId": 1026,
  "validationTime": 1620313234638,
  "expectationResults": [
    {
      "expectation": {
        "features": [
          "id"
        ],
        "rules": [
          {
            "level": "ERROR",
            "max": 0.0,
            "name": "HAS_DATATYPE"
          }
        ],
        "description": "validate inc_angle feature values",
        "name": "icebergs_id"
      },
      "results": [
        {
          "features": [
            "id"
          ],
          "message": "Success",
          "rule": {
            "level": "ERROR",
            "max": 0.0,
            "name": "HAS_DATATYPE"
          },
          "status": "SUCCESS",
          "value": "Distribution(Map(Boolean -> DistributionValue(0,0.0), Fractional -> DistributionV
0), String -> DistributionValue(1564,0.9750623441396509)),5)"
        }
      ],
      "status": "SUCCESS"
    },
    {
      "expectation": {
        "features": [
          "is_iceberg"
        ],
        "rules": [
          {
            "level": "ERROR",
            "min": 1.0,
            "name": "HAS_DATATYPE"
          },
          {
            "level": "ERROR",
            "max": 1.0,
            "min": 1.0,
            "name": "HAS_MAX"
          }
        ],
      }
    }
  ]
}
```

Figure 38. Feature validation results for an ingested dataframe

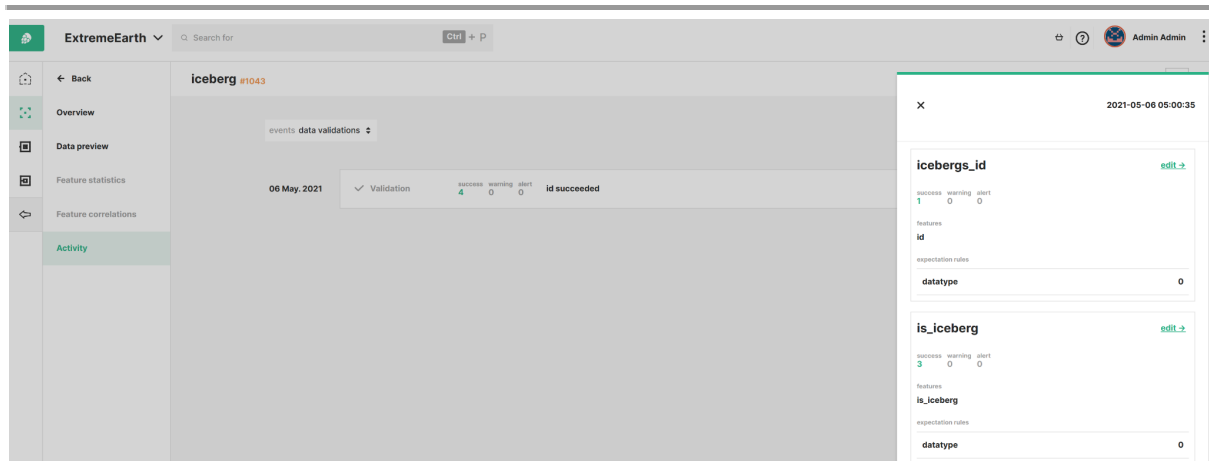


Figure 39. iceberg feature group validations in Hopsworks UI

8. Training

For doing machine learning training, it is useful to have a common abstraction that defines the type of training, the configuration parameters, the input dataset, and the infrastructure environment that the machine learning program runs in. In Hopsworks, the abstraction of an Experiment is used to encapsulate the aforementioned properties. To productionize ML models, it is important to be able to easily run a past experiment in case for example a software bug was discovered and the models need to be developed again based on previously seen data. A repeatable experiment is an abstraction that enables users to rerun a past experiment by managing to reproduce the execution environment, fetch the exact same data the original experiment ran on, and set the same configuration properties as well.

Section “Experiments & Distributed Training” of deliverable D1.1 describes initial work done on the Hopsworks Experiments framework. D1.1 focused on Experiments (section 7.1), Parallel Experiments (7.2), and Distributed Training (7.3). This deliverable builds on this work by demonstrating how to run all three types of experiments on the Iceberg dataset used by the previous stages of the Deep Learning pipeline.

Furthermore, the Experiments framework in Hopsworks has been extended with Maggy, a framework for distribution-transparent ML experiments, including distributed training, hyperparameter tuning, and ablation studies [19] [40] .

8.1 Experiments

Machine learning training was developed with TensorFlow version 2.4, an open-source end-to-end ML platform [47]. First, a training function needs to be defined that will be given as input into the Hopsworks experiments API that tracks the metadata of the training program and creates an instance of the Experiments abstraction. The training function first compiles the model that has been created in a previous cell of the notebook and then adds some callbacks for configuration of the TensorBoard. Then the Keras estimator, part of the Keras Deep open-source neural-network Python library [48], is used to train and evaluate the model. The input of the model is fetched from the training and test TFRecord datasets that were created from the feature store in the previous DL stage (section 6). After training is completed, the model is exported by using the serving module of the hops-util-py library of Hopsworks. Figure 40 shows the Jupyter configuration used to run the experiment.

Logs
JupyterLab

[Python](#)
[Experiments](#)
[Spark](#)

Experiments

Starting Jupyter with this mode will configure the PySpark Kernel.

☒ Experiment

☐ Parallel Experiments

☐ Distributed Training

Hours to shutdown:

Driver memory (MB):

Executor memory (MB):

Number of GPUs:

Advanced configuration ▾

Base Directory:

Fault-tolerant mode: ☐ OFF

+ Archive

No additional archives

+ Jar

No additional jars

+ Python

No additional python dependencies

+ File

No additional files

More Spark Properties:

Git: Disabled GitHub GitLab

Experiment

Run python wrapper functions on PySpark to run parallel hyperparameter optimization or distributed training orchestrated on PySpark executors.

The simple Experiment abstraction corresponds to a single Python experiment, for example any hyperparameters or other configuration is hard-coded in the code itself.

Want to learn more? See an [example](#) and [docs](#).

Documentation and resources

[readthedocs](#)

[hops python api](#)

[examples](#)

[github](#)

[website](#)

Accessing datasets >

Importing external modules >

Interact with filesystem >

Figure 40. Jupyter configuration for training the iceberg detection model

Figures 41-44 demonstrate the main parts of the training notebook, namely “train and evaluate”, exporting the model, and launching the experiment with the experiment API.

```
def create_model(input_shape):
    """Returns a CNN model for image classification.

    Parameters:
    - input_shape(tuple): input shape of the CNN model.

    Returns:
    - a TensorFlow keras model that is not compiled yet.

    """
    model = tf.keras.models.Sequential()

    # Conv Layer 1
    model.add(tf.keras.layers.Conv2D(64, kernel_size=(3, 3), activation='relu', input_shape=input_shape))
    model.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
    model.add(tf.keras.layers.Dropout(0.2))

    # Conv Layer 2
    model.add(tf.keras.layers.Conv2D(128, kernel_size=(3, 3), activation='relu' ))
    model.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
    model.add(tf.keras.layers.Dropout(0.2))

    # Conv Layer 3
    model.add(tf.keras.layers.Conv2D(128, kernel_size=(3, 3), activation='relu'))
    model.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
    model.add(tf.keras.layers.Dropout(0.2))

    # Conv Layer 4
    model.add(tf.keras.layers.Conv2D(64, kernel_size=(3, 3), activation='relu'))
    model.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
    model.add(tf.keras.layers.Dropout(0.2))

    # Flatten the data for upcoming dense layers
    model.add(tf.keras.layers.Flatten())

    # Dense Layers
    model.add(tf.keras.layers.Dense(512))
    model.add(tf.keras.layers.Activation('relu'))
    model.add(tf.keras.layers.Dropout(0.2))

    # Dense Layer 2
    model.add(tf.keras.layers.Dense(256))
    model.add(tf.keras.layers.Activation('relu'))
    model.add(tf.keras.layers.Dropout(0.2))

    # Sigmoid Layer
    model.add(tf.keras.layers.Dense(1))
    model.add(tf.keras.layers.Activation('sigmoid'))

    return model
```

Figure 41. Iceberg detection model architecture

```
# Training dataset in TFRecord format
train_ds = fs.get_training_dataset(name=TRAIN_FS_NAME).tf_data(target_name='is_iceberg')
train_ds = train_ds.tf_record_dataset(process=False, batch_size=TRAIN_BATCH_SIZE, num_epochs=EPOCHS)
train_ds_processed = train_ds.map(decode).shuffle(SHUFFLE_BUFFER_SIZE).repeat(EPOCHS).cache().batch(TRAIN_BATCH_SIZE).prefetch(tf.data.experimental.AUTOTUNE)

# Evaluation dataset in TFRecord format
eval_ds = fs.get_training_dataset(name=TEST_FS_NAME).tf_data(target_name='is_iceberg')
eval_ds = eval_ds.tf_record_dataset(process=False, batch_size=EVAL_BATCH_SIZE, num_epochs=EPOCHS)
eval_ds_processed = eval_ds.map(decode).shuffle(SHUFFLE_BUFFER_SIZE).repeat(EPOCHS).cache().batch(EVAL_BATCH_SIZE).prefetch(tf.data.experimental.AUTOTUNE)
```

Figure 42. Iceberg detection model training function (read training data)

```
# Start training the model.
history = model.fit(
    train_ds_processed,
    epochs=EPOCHS,
    verbose=1,
    validation_data=eval_ds_processed,
    callbacks=callbacks
)

# 'metrics' is the return value of this function;
# The values in 'metrics' will be printed to the notebook cell that launch the experiment
metrics = {
    'train_loss': history.history['loss'][-1],
    'train_accuracy': history.history['accuracy'][-1],
    'val_loss': history.history['val_loss'][-1],
    'val_accuracy': history.history['val_accuracy'][-1],
}

# ----- Training Process -----

# ----- Save and Export -----
# Export model as savedModel
export_path = tensorboard.logdir() + '/SavedModel'

tf.keras.models.save_model(
    model,
    export_path,
    overwrite=True,
    include_optimizer=True,
    save_format=None,
    signatures=None,
    options=None
)

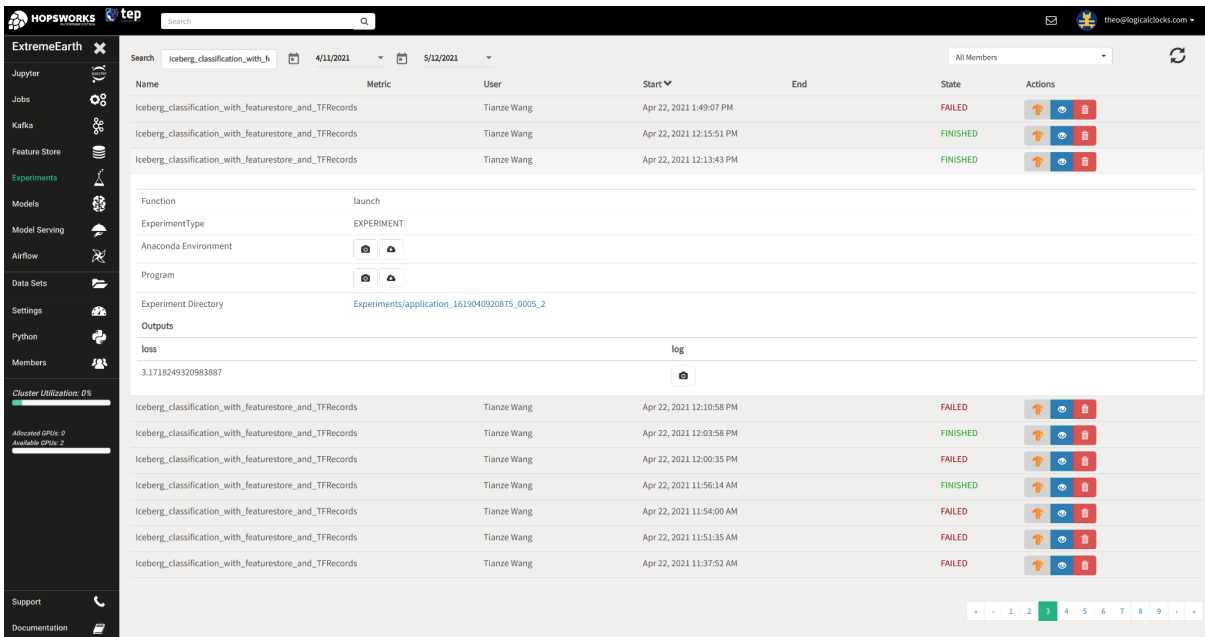
# 'hopsworks_model' is the module provided by hopsworks for exporting models
# 'hopsworks_model' is a different name of 'hops.model' to avoid name clashes
hopsworks_model.export(export_path, 'shipIcebergClassifier', metrics=metrics)
# ----- Save and Export -----










return metrics
```






Figure 43. Iceberg detection model training function, launch and export model

```
experiment.launch(train_fn, name='Iceberg_classification_with_featurestore_and_TFRecords', local_logdir=False)
Finished Experiment
({'hdfs://rpc namenode.service.consul:8020/Projects/ExtremeEarth/Experiments/application_1619040920875_0174_1', {'train_loss': 0.1419401615858078, 'train_accuracy': 0.949152827
2628784, 'val_loss': 15.787124633789062, 'val_accuracy': 0.8944281339645386, 'log': 'Experiments/application_1619040920875_0174_1/output.log'}}
```

Figure 44. Iceberg detection model experiments API launch training



Name	Metric	User	Start	End	State	Actions
Iceberg_classification_with_featurestore_and_TFRecords		Tianze Wang	Apr 22, 2021 1:49:07 PM		FAILED	  
Iceberg_classification_with_featurestore_and_TFRecords		Tianze Wang	Apr 22, 2021 12:15:51 PM		FINISHED	  
Iceberg_classification_with_featurestore_and_TFRecords		Tianze Wang	Apr 22, 2021 12:13:43 PM		FINISHED	  

Function	launch
ExperimentType	EXPERIMENT
Anaconda Environment	 
Program	 
Experiment Directory	Experiments/application_1619040920875_0005_2
Outputs	
loss	log
3.1718249320983887	






















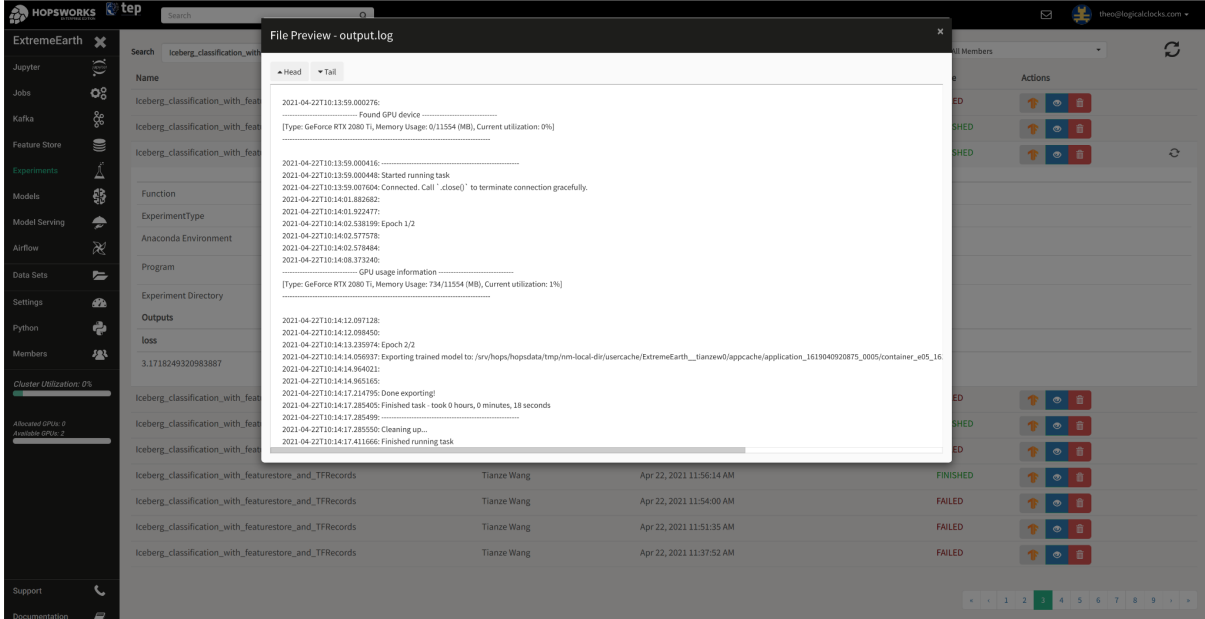












Name	User	Start	End	State	Actions
Iceberg_classification_with_featurestore_and_TFRecords	Tianze Wang	Apr 22, 2021 12:10:58 PM		FAILED	  
Iceberg_classification_with_featurestore_and_TFRecords	Tianze Wang	Apr 22, 2021 12:03:58 PM		FINISHED	  
Iceberg_classification_with_featurestore_and_TFRecords	Tianze Wang	Apr 22, 2021 12:00:35 PM		FAILED	  
Iceberg_classification_with_featurestore_and_TFRecords	Tianze Wang	Apr 22, 2021 11:56:14 AM		FINISHED	  
Iceberg_classification_with_featurestore_and_TFRecords	Tianze Wang	Apr 22, 2021 11:54:00 AM		FAILED	  
Iceberg_classification_with_featurestore_and_TFRecords	Tianze Wang	Apr 22, 2021 11:51:35 AM		FAILED	  
Iceberg_classification_with_featurestore_and_TFRecords	Tianze Wang	Apr 22, 2021 11:37:52 AM		FAILED	  

Figure 45. Experiment details in Hopworks Experiments registry



Name	User	Start	End	State	Actions
Iceberg_classification_with_featurestore_and_TFRecords	Tianze Wang	Apr 22, 2021 11:56:14 AM		FINISHED	  
Iceberg_classification_with_featurestore_and_TFRecords	Tianze Wang	Apr 22, 2021 11:54:00 AM		FAILED	  
Iceberg_classification_with_featurestore_and_TFRecords	Tianze Wang	Apr 22, 2021 11:51:35 AM		FAILED	  
Iceberg_classification_with_featurestore_and_TFRecords	Tianze Wang	Apr 22, 2021 11:37:52 AM		FAILED	  

File Preview - output.log

2021-04-22T10:13:59.000276: Found GPU device
[Type: GeForce RTX 2080 Ti, Memory Usage: 0/11554 (MB), Current utilization: 0%]

2021-04-22T10:13:59.000416: Started running task
2021-04-22T10:13:59.000448: Connected. Call 'close()' to terminate connection gracefully.
2021-04-22T10:14:01.885602: Epoch 1/2
2021-04-22T10:14:01.922477: Epoch 1/2
2021-04-22T10:14:02.538199: Epoch 1/2
2021-04-22T10:14:02.577578: Epoch 1/2
2021-04-22T10:14:02.579484: Epoch 1/2
2021-04-22T10:14:08.373240: GPU usage information
[Type: GeForce RTX 2080 Ti, Memory Usage: 734/11554 (MB), Current utilization: 1%]

2021-04-22T10:14:12.097128: Epoch 2/2
2021-04-22T10:14:12.098450: Epoch 2/2
2021-04-22T10:14:13.239704: Epoch 2/2
2021-04-22T10:14:14.056937: Exporting trained model to: /srv/hops/hopsdata/tmp/nm-local-dir/usercache/ExtremeEarth_tianzewb/opcache/application_1619040920875_0005/container_w05_16_2021-04-22T10:14:14.964021: Done exporting!
2021-04-22T10:14:14.965165: Done exporting!
2021-04-22T10:14:17.214795: Finished task - took 0 hours, 0 minutes, 18 seconds
2021-04-22T10:14:17.285495: Cleaning up...
2021-04-22T10:14:17.285556: Cleaning up...
2021-04-22T10:14:17.411666: Finished running task

Figure 46. Experiment logs shown from the Hopworks Experiments registry

8.2 Parallel Experiments

Parallel experiments can significantly speed up the process of exploring hyper-parameter combinations that work best for the ML model. Hopworks Experiment API makes hyper-parameter search trivial, by allowing users to define the search space in a dictionary which is provided as input into the same `experiment.launch` method demonstrated in

Section 7.1. The rest of the notebook, including the training function and exporting the model, is the same as the single experiment one. Figures 47 and 48 demonstrate how to set different learning rates and feed them as input into the notebook.

```
args_dict = {'learning_rate': [0.001, 0.0005, 0.0001, 0.005, 0.003, 0.009]}
```

```
experiment.launch(train_fn, args_dict, name="Iceberg_classification_with_Parallel_Training", local_logdir=True)
```

Finished Experiment

```
('hdfs://rpc.namenode.service.consul:8020/Projects/ExtremeEarth/Experiments/application_1619040920875_0054_1', None)
```

Figure 47. Iceberg hyper parameter tuning with parallel experiments














Iceberg_classification_with_Parallel_Training		Tianze Wang		Apr 23, 2021 12:33:42 PM	Apr 23, 2021 12:39:00 PM	FINISHED	  	
Function	launch							
ExperimentType	EXPERIMENT							
Anaconda Environment	<div> </div>							
Program	<div> </div>							
Experiment Directory	Experiments/application_1619040920875_0054_1							
Model	Models/ship_iceberg_classifier/9							
Parameters	Outputs							
learning_rate	val_loss	log	train_loss	train_accuracy	val_accuracy			
0.0001	0.48354488611221313		0.5344764590263367	0.6912114024162292	0.7771260738372803			
0.0005	0.3290237784385681		0.2499350756406784	0.8864607810974121	0.8533724546432495			
0.001	0.32901686429977417		0.2715916037559509	0.8774346709251404	0.8416422009468079			
0.003	0.37866631150245667		0.3936786651611328	0.8272367119789124	0.8357771039009094			
0.005	0.3994578719139099		0.45547741651535034	0.7680126428604126	0.7976539731025696			
0.009	0.6926142573356628		0.6882401704788208	0.5334916710853577	0.5190615653991699			

Figure 48. Iceberg hyper parameter tuning with parallel experiments in the Hopsworks Experiments registry

While training is ongoing, users can follow the progress of the parallel experiments by navigating to the TensorBoard of this experiment via the Hopsworks Experiments service. Figure 49 shows the tensorboard of the parallel experiments notebook.

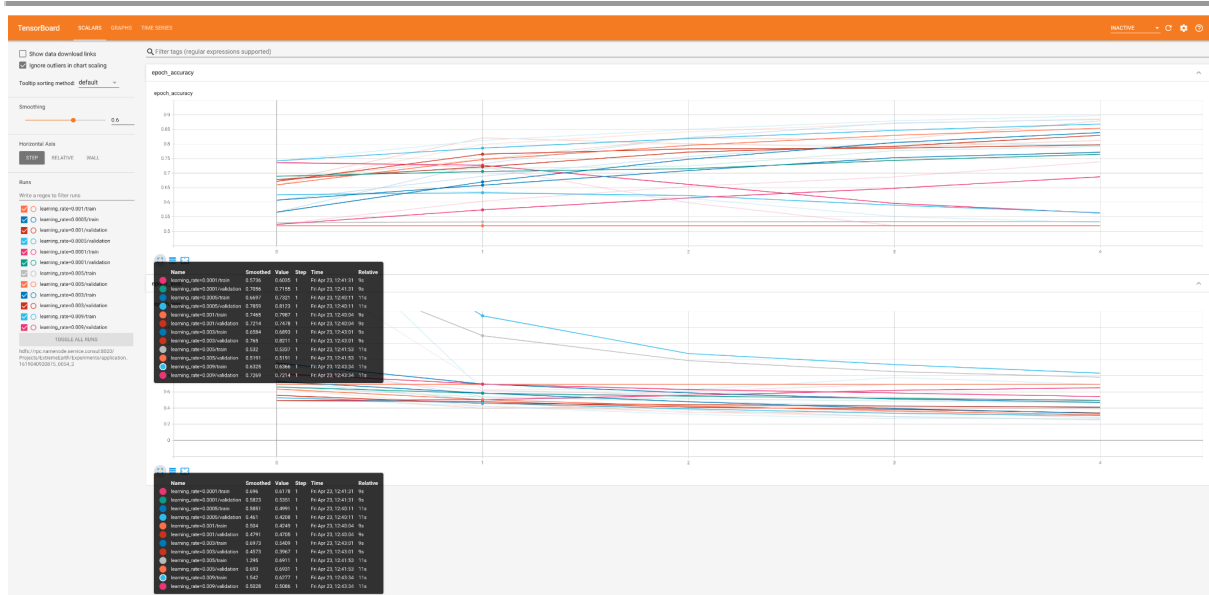


Figure 49 . Iceberg hyper parameter tuning with parallel experiments tensorboard

Jupyter needs to be launched with the Parallel Experiments configuration to enable running notebooks with the parallel experiments API. Figure 50 shows how Jupyter was started for this running the iceberg parallel experiments.

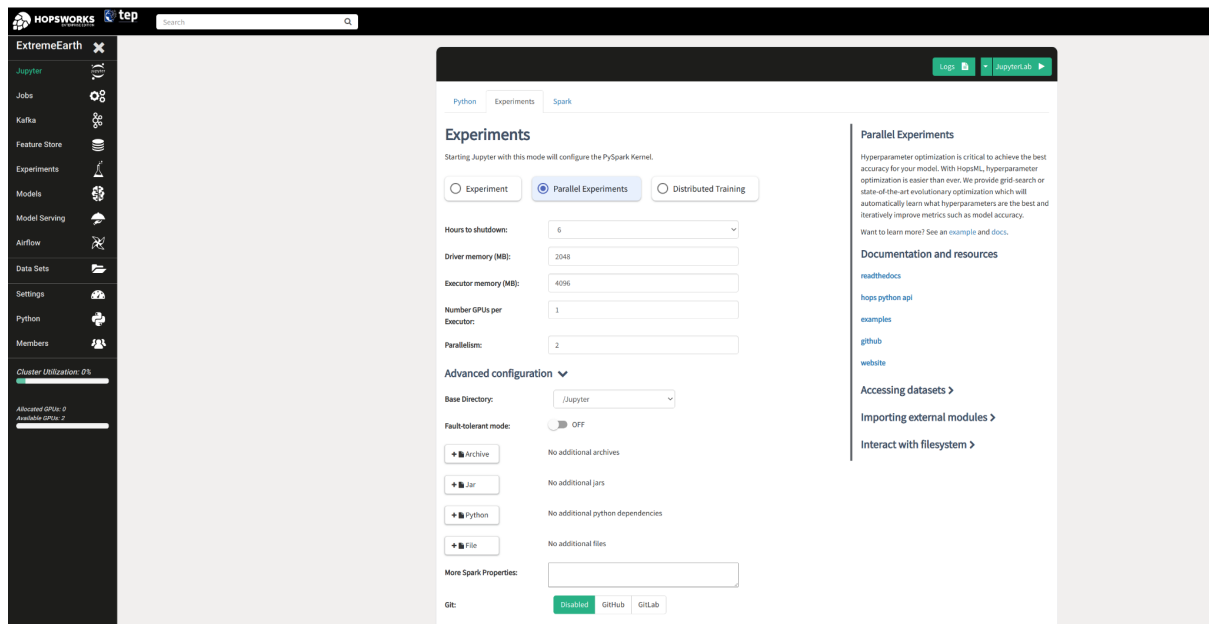


Figure 50. Jupyter configuration for parallel experiments with GPUs

8.3 Distributed Training

For distributed training, the same model was used as in the previous sections, however, Jupyter was started with the Distributed Training configuration. In particular, the Mirrored Strategy as shown in Figure 51. Figures 52 and 53 demonstrate how the experiments API is used for distributing training from a Jupyter notebook. Details of the architecture and implementation of the distributed training experiments framework in Hopsworks is available in the deliverable D1.5 Hops data platform integration guide for applications - version II.

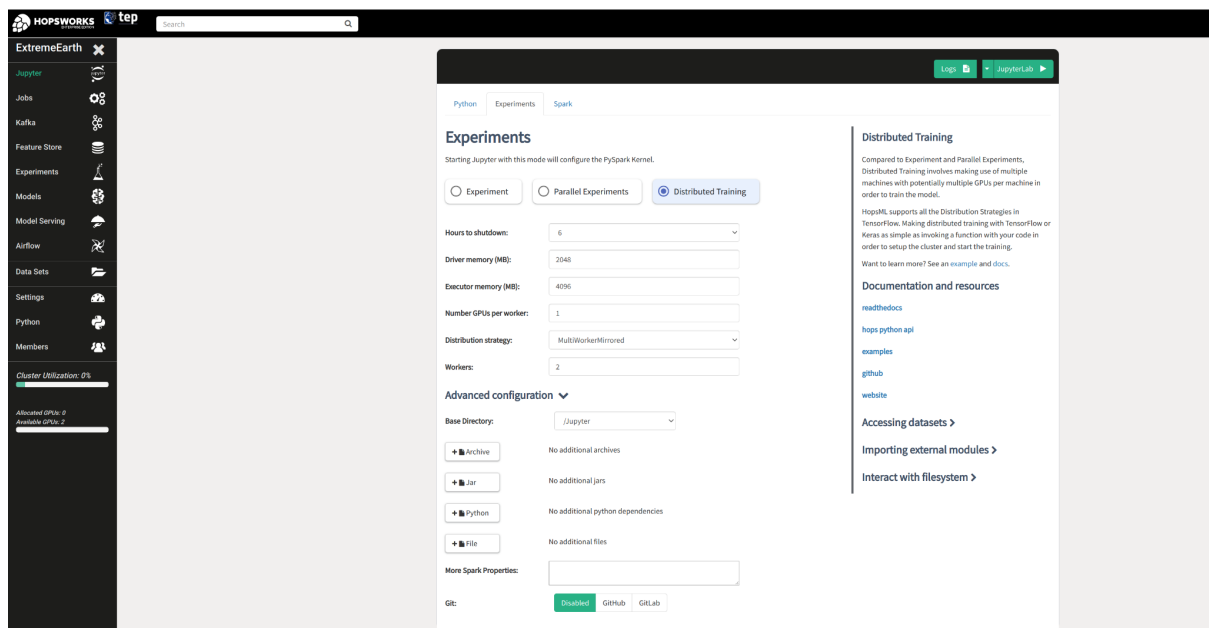


Figure 51. Distributed training Jupyter configuration. 2 workers with 1 GPU Each

```
# ----- Training Process -----

# ----- Save and Export -----
# Export model as savedModel
# export_path = tensorboard.logdir() + '/SavedModel'
export_path = os.getcwd() + '/SavedModel'

tf.keras.models.save_model(
    model,
    export_path,
    overwrite=True,
    include_optimizer=True,
    save_format=None,
    signatures=None,
    options=None
)

# 'hopsworks_model' is the module provided by hopsworks for exporting models
# 'hopsworks_model' is a different name of 'hops.model' to avoid name clashes

# Only need to export the model on the chief
if json.loads(os.environ['TF_CONFIG'])['task']['type'] == 'chief':
    hopsworks_model.export(export_path, 'ship_iceberg_classifier', metrics=metrics)

return metrics
```

Figure 52. Iceberg distributed training function

```
experiment.mirrored(train_fn, name='Iceberg_Ship_Classification_with_distributed_training', metric_key='val_accuracy')
Finished Experiment
{'hdfs://rpc namenode.service.consul:8020/Projects/ExtremeEarth/Experiments/application_1619040920875_0162_7', {'train_loss': 0.29985445737838745, 'train_accuracy': 0.86603325
60539246, 'val_loss': 0.2837695777416229, 'val_accuracy': 0.8533724546432495, 'log': 'Experiments/application_1619040920875_0162_7/chief_0_output.log'}}
```

Figure 53. Iceberg distributed training experiments API launch

The notebook itself for distributed training is available online in the ExtremeEarth GitHub repository [34].

8.4 Hyperparameter Tuning with Maggy

Hopsworks has been extended with a new framework called Maggy for performing efficient asynchronous optimization of expensive black-box functions on top of Apache Spark. Maggy is not bound to stage-based optimization algorithms, contrary to existing frameworks. Therefore it is able to make extensive use of early stopping in order to achieve efficient resource utilization [19]. As of this deliverable, Maggy supports asynchronous hyperparameter tuning of machine learning and deep learning models, and ablation studies on neural network layers as well as input features.

The main component of Maggy is an RPC mechanism that is implemented that enables results of trials to be reported from the executors back to the driver in Spark. The Optimizer

component that runs on the driver is then responsible for deciding when to stop a trial and send new tasks to executors. The latter are blocked by long-running tasks so they can run multiple trials for every scheduled task, instead of only one trial per task as was the case previously.

Figure 54 depicts the RPC mechanism implemented between the driver and executors in Maggy.



Figure 54. Maggy early stopping in Apache Spark

Creating and evaluating the model is similar to the previous experiments examples. The training function *train_fn* in the iceberg Maggy notebook optimizes three hyper-parameters, kernel, pool, and dropout. Figure 55 demonstrates how the Maggy reporter is configured in the training function.

```
def train_fn(kernel, pool, dropout, reporter):
    """Wrapper function for the experiment.

    Parameters:
    - learning_rate: learning rate of the optimizer during training.

    Returns:
    - metrics: training summary.

    """

    # ----- Initialization -----
    # Establish a connection with the Hopsworks feature store
    # engine='training' is needed so that the executors in Spark can connect to feature store
    connection = hsfs.connection(engine='training')
    # Get the feature store handle for the project's feature store
    fs = connection.get_feature_store()

    # Clear session info
    tf.keras.backend.clear_session()

    # Set up visible GPU
    gpus = tf.config.experimental.list_physical_devices('GPU')
    if gpus:
        try:
            # Currently, memory growth needs to be the same across GPUs
            for gpu in gpus:
                tf.config.experimental.set_memory_growth(gpu, True)
            logical_gpus = tf.config.experimental.list_logical_devices('GPU')
            print(len(gpus), "Physical GPUs,", len(logical_gpus), "Logical GPUs")
        except RuntimeError as e:
            # Memory growth must be set before GPUs have been initialized
            print(e)

    # ----- Initialization -----

    # ----- Hyperparameters -----
    # Number of epochs to training
    EPOCHS = 10 # as we are limited with CPU for demo
    # Training batch size
    TRAIN_BATCH_SIZE = 32
    # Evaluation batch size
    EVAL_BATCH_SIZE = 1
    # Shuffle buffer size for TensorFlow dataset
    SHUFFLE_BUFFER_SIZE = 10000
    # learning rate of the optimizer during training
    LEARNING_RATE = 0.001
    # input_shape of the model
    INPUT_SHAPE = (75, 75, 3)
    # Name of the training dataset in feature store
    TRAIN_FS_NAME = 'train_tfrecords_iceberg_classification_dataset'
    # Name of the test dataset in feature store
    TEST_FS_NAME = 'test_tfrecords_iceberg_classification_dataset'
```

Figure 55. Iceberg hyper-parameter optimization with Maggy - training function

Figures 56 and 57 demonstrate how the search space for hyper-parameters is defined with Maggy and then how the experiment is launched by using the `experiment.lagom` API. The output of the `experiment.lagom` invocation is printed in the notebook itself under the cell that starts the experiment, and a progress bar gets updated as the trials finish executing.

Also, the print function is overridden to redirect the output from the workers output to the cell output, which makes debugging and experimentation easier.

```
from maggy import Searchspace

# The searchspace can be instantiated with parameters
sp = Searchspace()

# Or additional parameters can be added one by one
sp.add('kernel', ('INTEGER', [3, 4]))
sp.add('pool', ('INTEGER', [2, 3]))
sp.add('dropout', ('DOUBLE', [0.10, 0.50]))

Hyperparameter added: kernel
Hyperparameter added: pool
Hyperparameter added: dropout
```

Figure 56. Iceberg hyper-parameter optimization with Maggy - search space

Launch the hyperparameter optimization

```
[ ]: from maggy import experiment
    from maggy.experiment_config import OptimizationConfig

[7]: config = OptimizationConfig(
        num_trials=10,
        optimizer='randomsearch',
        searchspace=sp,
        direction='max',
        es_interval=1,
        es_min=2,
        hb_interval=5,
        name='Iceberg_Classification_Maggy'
    )

[8]: result = experiment.lagom(train_fn=train_fn, config=config)

HBox(children=(FloatProgress(value=0.0, description='Maggy experiment', max=10.0, style=ProgressStyle(descript...
0: Connected. Call `.close()` to terminate connection gracefully.
0: Physical devices cannot be modified after being initialized
1: Connected. Call `.close()` to terminate connection gracefully.
1: Physical devices cannot be modified after being initialized
1:
1:
1: Epoch 1/10
1:
0:
0:
0: Epoch 1/10
0:
0:
1:
1:
0:
0:
1: Epoch 2/10
0: Epoch 2/10
1: Epoch 3/10
0: Epoch 3/10
1: Epoch 4/10
0: Epoch 4/10
1: Epoch 5/10
1: Epoch 6/10
```

Figure 57. Iceberg hyper-parameter optimization with Maggy - launch

Once all trials are executed, a summary of results is printed as the final output, as can be seen in Figure 58.

```
0: Epoch 10/10
You are running Maggy on Hopsworks.

----- RandomSearch Results ----- direction(max)
BEST combination {"kernel": 4, "pool": 3, "dropout": 0.14127066071482483} -- metric 0.5446287393569946
WORST combination {"kernel": 4, "pool": 3, "dropout": 0.4702859921298028} -- metric 0.2919609844684601
AVERAGE metric -- 0.35823622047901155
EARLY STOPPED Trials -- 0
Total job time 0 hours, 21 minutes, 1 seconds

Finished experiment.
```

Figure 58. Iceberg hyper-parameter optimization with Maggy - results

8.5 Ablation studies

In the context of machine learning, we can define an ablation study as “a scientific examination of a machine learning system by removing its building blocks in order to gain insight on their effects on its overall performance”. Dataset features and model components are notable examples of these building blocks (hence we use their corresponding terms of feature ablation and model ablation), but any design choice or module of the system may be included in an ablation study. By removing each building block (e.g., a particular layer of the network architecture, or a set of features of the training dataset), retraining, and observing the resulting performance, we can gain insights into the relative contributions of each of these building blocks.

An ablation study can be thought of as an experiment that consists of several trials. For example, each model ablation trial involves training a model with one or more of its components (e.g., a layer) removed. Similarly, a feature ablation trial involves training a model using a different set of dataset features, and observing the outcomes.

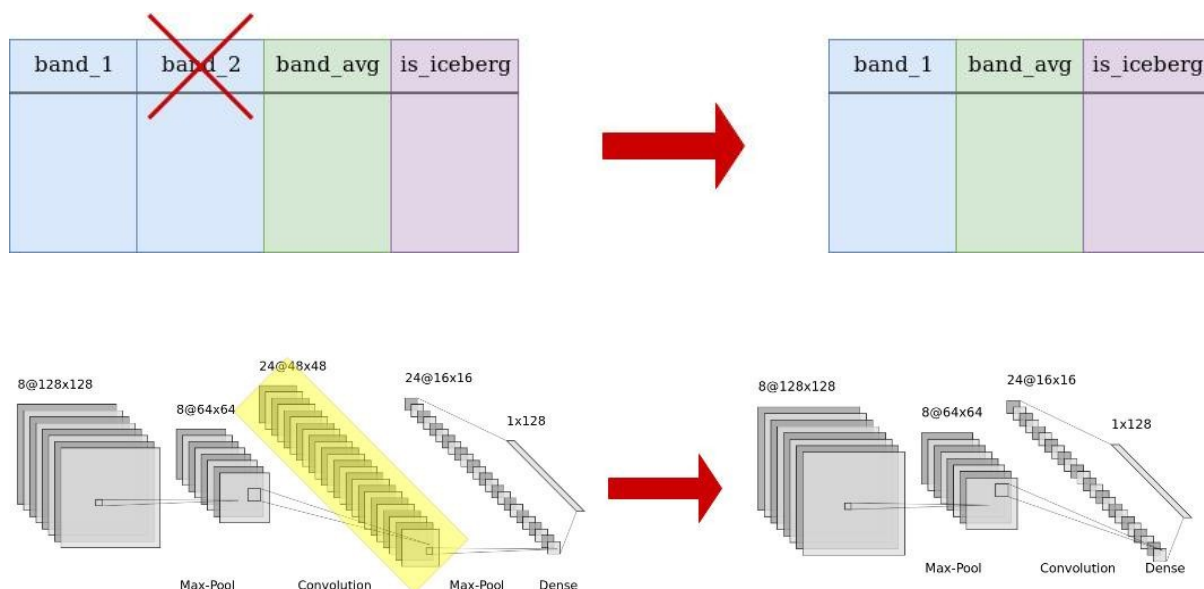


Figure 59. Ablation studies architecture

With Maggy, performing ablation studies of machine learning or deep learning systems is a fairly simple task that consists of the following steps:

- Creating an `AblationStudy` instance,
- Specifying the components that you want to ablate by including them in your `AblationStudy` instance,
- Defining a base model generator function and/or a dataset generator function,
- Wrapping your TensorFlow/Keras code in a Python function (called e.g., the training function) that receives two arguments (`model_function` and `dataset_function`), and
- Launching your experiment with Maggy while specifying an ablation policy.

Maggy will then take care of generating the corresponding ablation trials and executing them in parallel.

```
from maggy.ablation import AblationStudy

# create an AblationStudy instance
iceberg_ablation = AblationStudy('iceberg', training_dataset_version=1, label_name="is_iceberg")

# pass the model generator function to ablation study
iceberg_ablation.set_dataset_generator(create_datasets)
# set the base model generator
iceberg_ablation.model.set_base_model_generator(create_model)

# add layers to the ablation study
iceberg_ablation.model.layers.include(['my_conv_1', 'my_conv_2', 'my_conv_3', 'my_conv_4'])
iceberg_ablation.model.layers.include(['my_maxpool_1', 'my_maxpool_2', 'my_maxpool_3', 'my_maxpool_4'])
iceberg_ablation.model.layers.include(['my_dropout_1', 'my_dropout_2', 'my_dropout_3', 'my_dropout_4'])

iceberg_ablation.model.layers.print_all()

# add a layer group using a prefix

# iceberg_ablation.model.layers.include_groups(prefix='my_conv')
# iceberg_ablation.model.layers.include_groups(prefix='my_maxpool')
# iceberg_ablation.model.layers.include_groups(prefix='my_dropout')

# iceberg_ablation.model.layers.print_all_groups()

print('\n\nAblation Study summary: \n {}'.format(iceberg_ablation.to_dict()))

Included single layers are:

my_conv_1
my_dropout_1
my_conv_2
my_conv_3
my_maxpool_2
my_maxpool_1
my_maxpool_4
my_dropout_2
my_maxpool_3
my_conv_4
my_dropout_4
my_dropout_3

Ablation Study summary:
{'training_dataset_name': 'iceberg', 'training_dataset_version': 1, 'label_name': 'is_iceberg', 'included_features': [], 'included_layers': ['my_conv_1', 'my_dropout_1', 'my_conv_2', 'my_conv_3', 'my_maxpool_2', 'my_maxpool_1', 'my_maxpool_4', 'my_dropout_2', 'my_maxpool_3', 'my_conv_4', 'my_dropout_4', 'my_dropout_3'], 'custom_dataset_generator': True}
```

Figure 60. Maggy ablation studies notebook example - ablations

The above figure shows the process of defining a model ablation study experiment with Maggy. Users can include individual layers, or “layer groups”, e.g., for blocks of similar layers generated with loops.

After the training function is defined, the user has to pass it to Maggy’s `lagom` method to launch the experiment in parallel (in case there are multiple workers):

```
# Create a config for lagom
from maggy.experiment_config import AblationConfig

config = AblationConfig(name='Iceberg_ship_classifier_ablation_study', ablation_study=iceberg_ablation, ablator='loco', description='Abla

from maggy import experiment

# launch the experiment
result = experiment.lagom(train_fn=train_fn, config=config)

0: Epoch 18/20
0: Epoch 19/20
0: Epoch 20/20
Warning: deepspeed and/or fairscale import failed. DeepSpeed backend and zero_lvl 3
won't be available
You are running Maggy on Hopsworks.

----- LOCO Results -----
BEST Config Excludes {"ablated feature": "None", "ablated layer": "my_conv_3"} -- metric 0.8944281339645386
WORST Config Excludes {"ablated feature": "None", "ablated layer": "my_conv_1"} -- metric 0.8416422009468079
AVERAGE metric -- 0.877960741519928
Total Job Time 1 hours, 32 minutes, 58 seconds

Finished experiment.
```

Figure 61. Maggy ablation studies notebook example - results

9. Model Analysis

Section 3.3 of deliverable D1.5 Hops data platform integration guide for applications - version II describes how Hopsworks users can use the What-If tool for doing model analysis on Hopsworks. This demonstrator deliverable shows how this tool can be used to perform model analysis on the demonstrator dataset. Deliverable D1.4, the previous version of this deliverable, integrated TensorFlow Model Analysis (TFMA) with Apache Beam and Apache Flink. This deliverable does not make use of this integration as it could be overly complex for the majority of cases where model analysis needs to be done. Also, the tools involved have frequent braking changes and are focused on execution in the Google cloud environment. Users can still utilize the Python and Flink support in Hopsworks to use such tools if needed as. The What-If tool covers the great majority of use cases and also fits seamlessly into the current demonstrator that uses TensorFlow for model development. In addition, the What-If tool provides powerful and interactive visualizations via Jupyter.

Hopsworks has been expanded to include the What-If as part of the default Python environments that projects in Hopsworks come with. Therefore, users do not need to install it separately avoiding any risks of Python library dependency conflicts as well.

Figure 62 shows the code snippet used to perform model analysis for the sea iceberg classification model developed with the demonstrator dataset in this deliverable. Users set the number of data points to be displayed, the test dataset location to be used for analysis of the model, and the features to be used.

```
##@title Invoke What-If Tool for test data and the trained model {display-mode: "form"}

num_datapoints = 2000 #@param {type: "number"}
tool_height_in_px = 1000 #@param {type: "number"}

from witwidget.notebook.visualization import WitConfigBuilder
from witwidget.notebook.visualization import WitWidget

test_examples = df_to_examples(test_df, features_and_labels)

# Setup the tool with the test examples and the trained classifier
config_builder = WitConfigBuilder(test_examples).set_estimator_and_feature_spec(classifier, feature_spec).set_label_vocab(['not iceberg', 'is iceberg'])
WitWidget(config_builder, height=tool_height_in_px)
```

Figure 62. Model analysis what-if tool code snippet

Figure 63 depicts the performance and fairness of the model based on a particular feature of the model. Figure 64 shows descriptive statistics for the feature spec provided to the what-if tool.

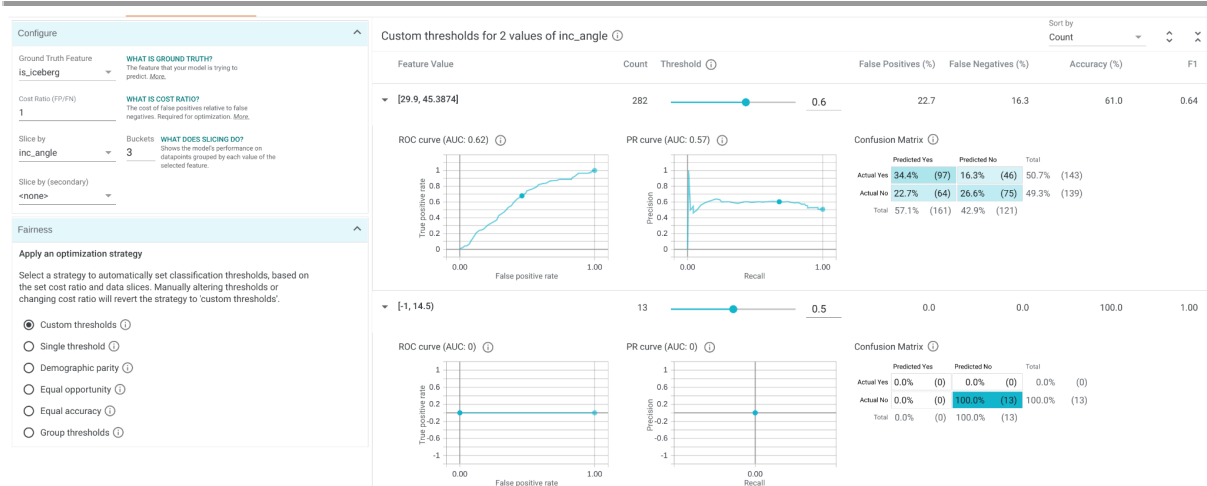


Figure 63. Performance and Fairness of the model

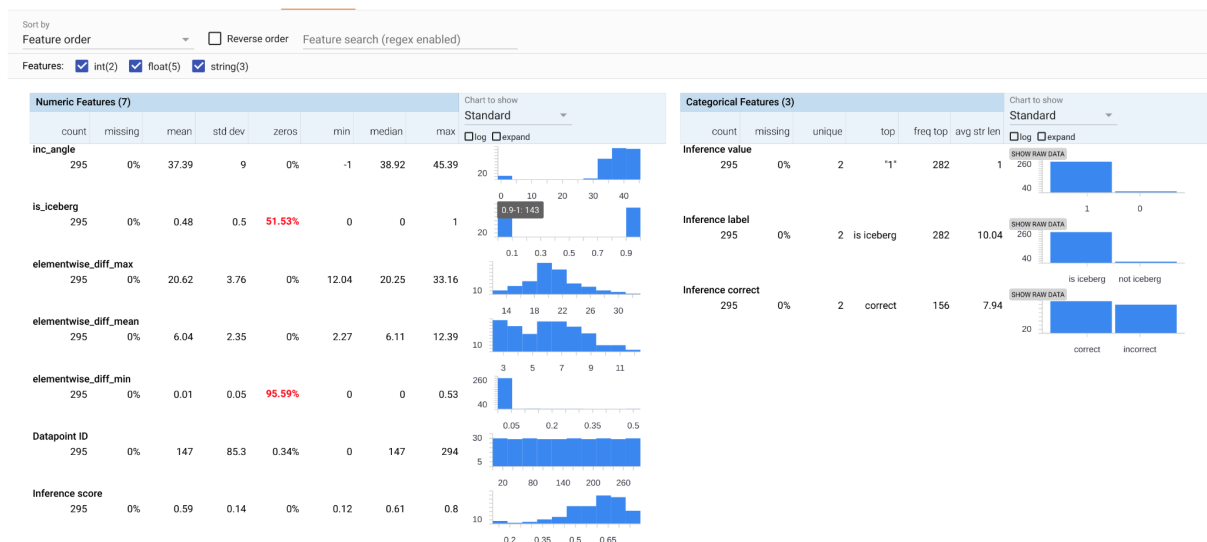


Figure 64. Feature statistics

10. Model Serving & Monitoring

Deliverable 1.1 provided design guidelines for productionizing model serving with Hopsworks. After a model has been developed and exported by the previous stages in the DL pipeline, it needs to be served so that external clients can use it for inference.

After the model is deployed, its performance needs to be monitored in real-time so that users can decide when it would be the best time to trigger the training stage. Hopsworks has been extended to provide support for TensorFlow serving and Scikit-learn, an open-source ML Python library [51]. Hopsworks has been extended with a Kubernetes cluster on which docker containers are deployed that run TensorFlow serving and Scikit-learn. Users have the option to select the number of instances for model serving at runtime, therefore Hopsworks provides users with the important property of elasticity.

This demo uses TensorFlow serving as the model has been developed and exported using TensorFlow. Inference requests are proxied through the Hopsworks REST API to provide secure multi-tenant access to Hopsworks where role-based access control is done based on projects. Project members are allowed to submit requests only to the models being served from within their projects.

Inference requests are logged in Apache Kafka [20] which is provided as a multi-tenant service in Hopsworks. Avro schemas [21] are attached to Apache Kafka topics in Hopsworks. By default, each project gets a default inference schema, depicted in Figure 65.

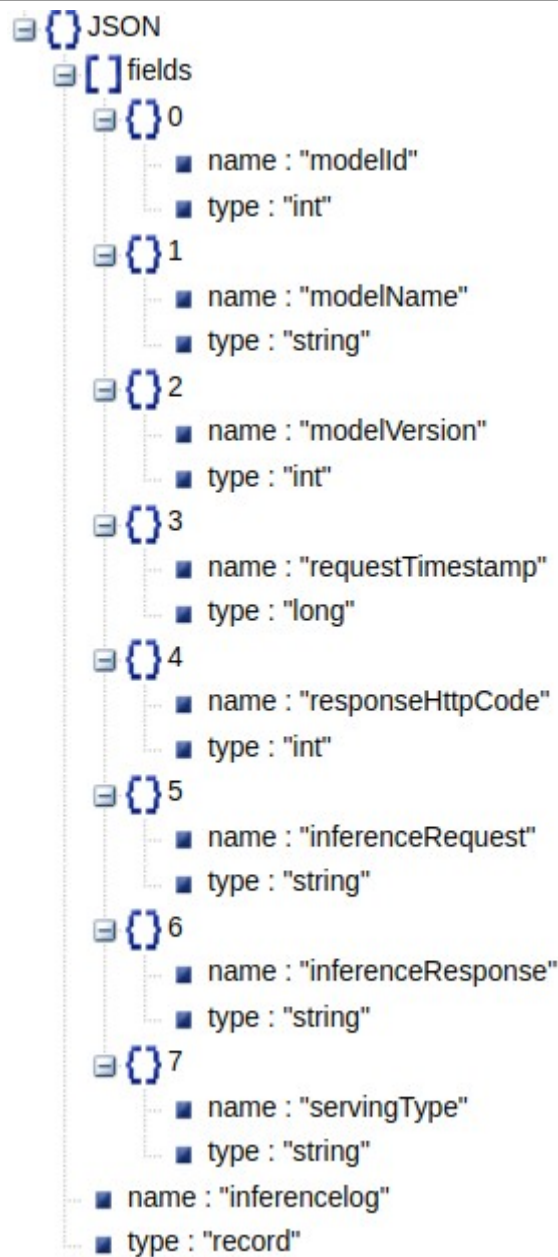


Figure 65. Inference avro schema

The schema is used to store the inference requests in Apache Kafka in a structured way, so that client applications can then read in real-time the inference requests and apply some business logic on how the model is performing. Figure 66 depicts the overall architecture of model serving and monitoring in Hopsworks.

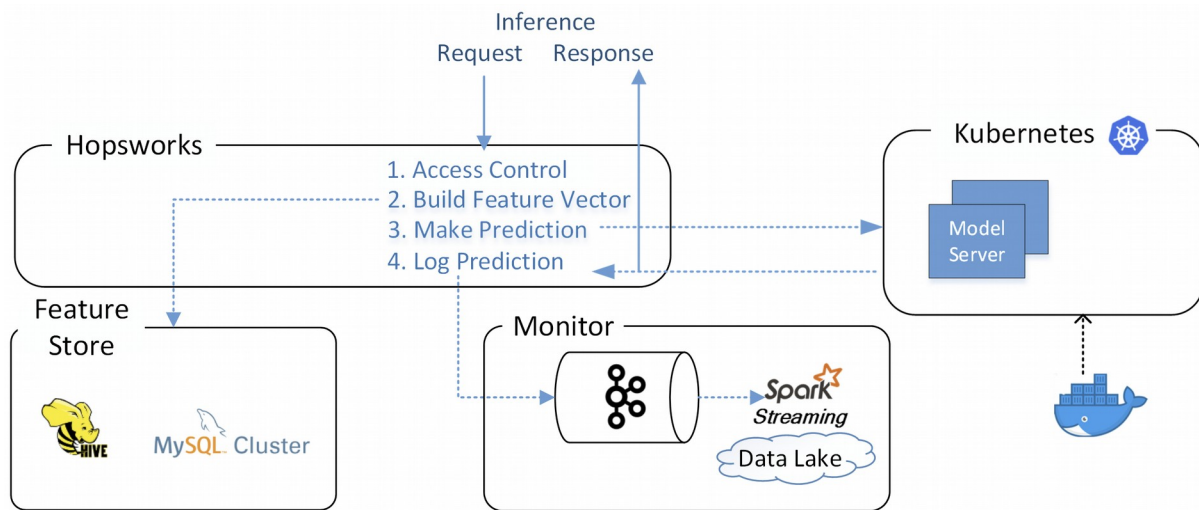


Figure 66. Model monitoring and logging architecture

In the previous Training stage of the DL pipeline, the model was exported by using the serving module of the hops-util-py library. The model is persisted under the dataset Models and the name chosen for this demo was *ship_iceberg_classifier*. Figure 67 demonstrates how the model serving instance for *ship_iceberg_classifier* is created. The fields users can set are:

- Model: The directory where the versions of the model are stored. The directory structure respects the TensorFlow serving directory convention.
- Model Version: Which version of the model to be served.
- Request batching: Whether to batch inference requests.
- Instances: Number of model serving instances to be spawned in Kubernetes.
- Kafka topic: Whether to create a new Kafka topic to store the inference requests.
- Kafka Num Partitions: Number of topic partitions.
- Kafka replication factor: The replication factor for each topic.

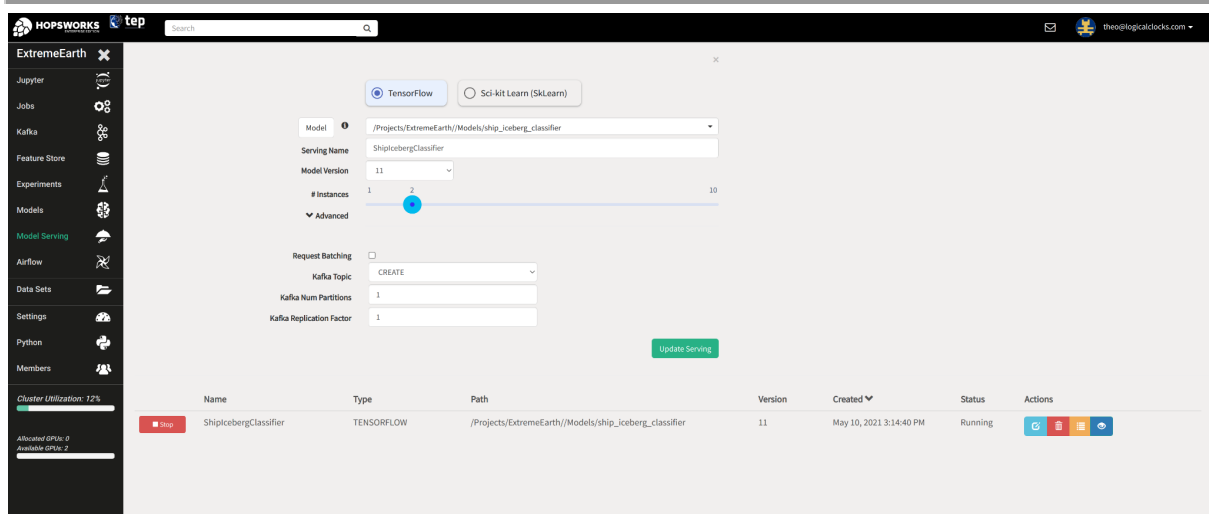


Figure 67. Model serving create UI

Figure 68 shows the main Model Serving dashboard after the serving instance has started. By clicking the “Show Detailed Information” button, users can view the endpoints where inference requests are being served from.

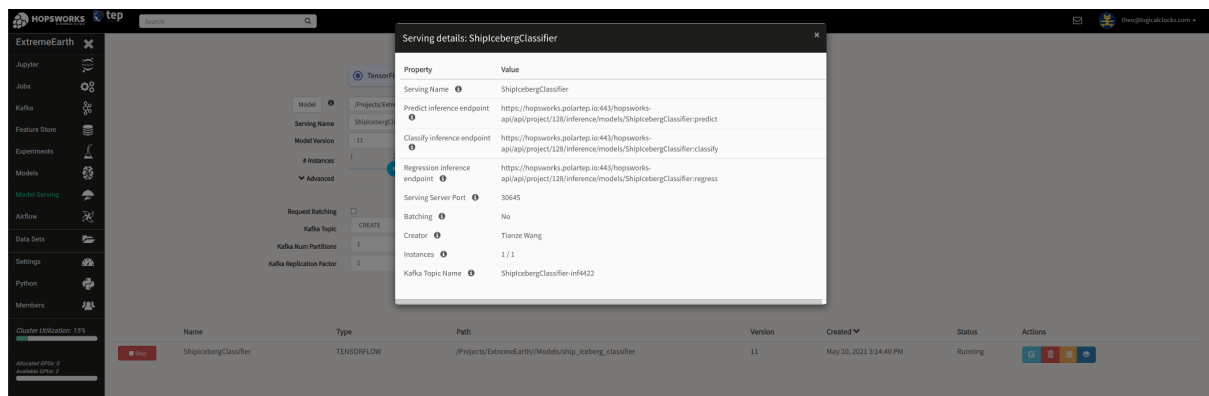


Figure 68. Model serving details

An important aspect of making a model serving production-ready, is to be able to collect logs in real-time and make them easily accessible to users. Hopsworks uses the ELK stack to achieve that, as it collects logs using Filebeat, persists them in Elasticsearch, and visualizes them with Kibana. Figure 69 shows the logs of *ship_iceberg_classifier* TensorFlow serving instances.



Platform Software Architecture - Version II, D1.6

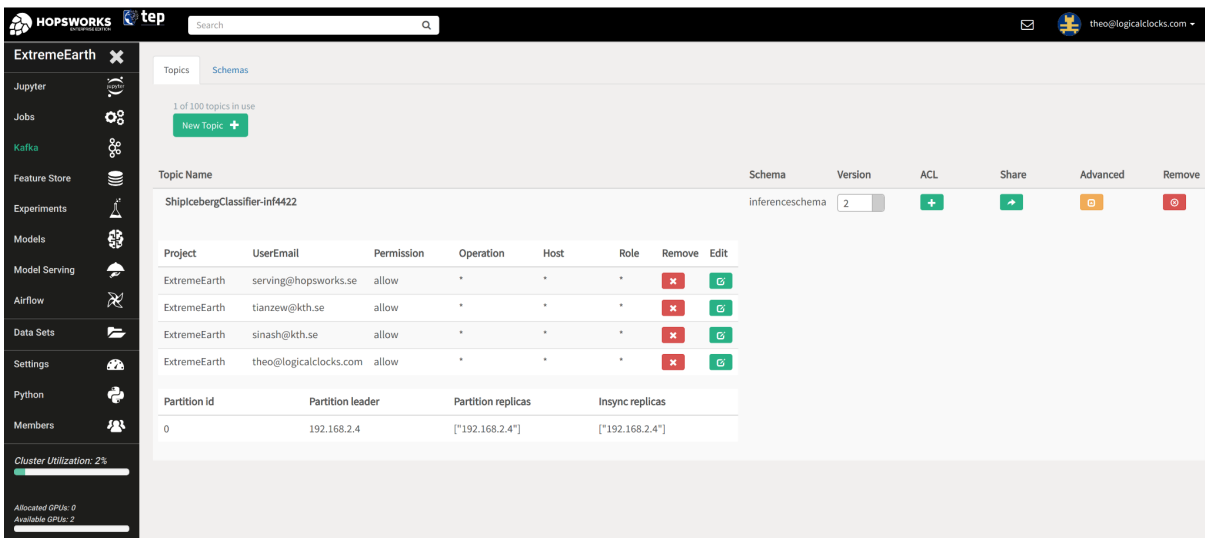
```
[4]: %local
def do_inference():
    with tf.Session() as sess:
        dataset = get_tf_dataset()
        dataset_iter = dataset.make_one_shot_iterator()
        next_element = dataset_iter.get_next()
        for i in range(10):
            x,y = sess.run(next_element)
            request_data={}
            request_data['instances'] = [x.tolist()]
            response = serving.make_inference_request("icebergmodel", data=request_data, verb= ":predict")
            print("prediction: {}, is_iceberg: {}".format(response['predictions'][0][0], y[0]))

[7]: %local
do_inference()

prediction: 0.457645357, is_iceberg: 0.0
prediction: 0.457955331, is_iceberg: 0.0
prediction: 0.469079971, is_iceberg: 0.0
prediction: 0.458596766, is_iceberg: 1.0
prediction: 0.458393067, is_iceberg: 1.0
prediction: 0.459837317, is_iceberg: 1.0
prediction: 0.456256419, is_iceberg: 0.0
prediction: 0.457359701, is_iceberg: 0.0
prediction: 0.457865953, is_iceberg: 0.0
prediction: 0.458710402, is_iceberg: 0.0
```

Figure 70. Submitting inference requests with the Hopsworks Python client APIs

In this demo, 10 inference requests were submitted. In another Python program from an IPython notebook, the monitoring job is started. The first step is to connect to the *ShiplcebergClassifier-inf4422* Kafka topic that stores the inference requests and their metadata. Users can also manage the topic from the Kafka service menu in Hopsworks UI, as depicted in Figure 71.



The screenshot shows the Hopsworks UI interface. On the left is a sidebar with navigation options: ExtremeEarth, Jupyter, Jobs, Kafka, Feature Store, Experiments, Models, Model Serving, Airflow, Data Sets, Settings, Python, and Members. The main panel displays the 'Topics' section for 'ShiplcebergClassifier-inf4422'. It shows a table with columns: Project, UserEmail, Permission, Operation, Host, Role, Remove, and Edit. Below this is a table for 'Partition id' with columns: Partition id, Partition leader, Partition replicas, and Insync replicas. The bottom status bar indicates 'Cluster Utilization: 2%' and 'Allocated GPUs: 0 Available GPUs: 2'.

Project	UserEmail	Permission	Operation	Host	Role	Remove	Edit
ExtremeEarth	serving@hopsworks.se	allow	*	*	*		
ExtremeEarth	tianzew@kth.se	allow	*	*	*		
ExtremeEarth	sinash@kth.se	allow	*	*	*		
ExtremeEarth	theo@logicalclocks.com	allow	*	*	*		

Partition id	Partition leader	Partition replicas	Insync replicas
0	192.168.2.4	["192.168.2.4"]	["192.168.2.4"]

Figure 71. Model inference logging Kafka topic details

In the notebook, the Kafka client (consumer) is instantiated and subscribes to the inference topic as shown in Figure 72.

```
consumer = Consumer(config)
def print_assignment(consumer, partitions):
    """
    Callback called when a Kafka consumer is assigned to a partition
    """
    print('Assignment:', partitions)
topics = [TOPIC_NAME]
consumer.subscribe(topics, on_assign=print_assignment)
```

Figure 72. Submitting inference requests with the Hopsworks Python client APIs

For brevity, in this example one message is consumed from the topic, that is one inference request, and part of its data is logged, as shown in Figure 73. It is trivial to change the number of inference requests to be logged periodically by modifying the logging loop.

```
message = {}
for i in range(0, 1):
    msg = consumer.poll(timeout=5.0)
    if msg is not None:
        message = msg
        print('Consumed Message: {} from topic: {}, partition: {}, offset: {}, timestamp: {}'.format(msg.value(), msg.topic(), msg.partition(), msg.offset(), msg.timestamp()))
    else:
        print("Topic empty, timeout when trying to consume message, try to produce messages to the topic and then re-consume")
```

Consumed Message: b'\x02\x18icebergmodel\x04\xb0\xal\xc9\xb2\xd6[\x90\x03\xe4\xe6+["instances": [[[-32.045310974121094, -25.196855545043945, -28.621084213256836], [-30.461740493774414, -26.93991470336914, -28.70082664489746], [-28.332691192626953, -27.608448028564453, -27.970569610595703], [-32.04547882080078, -29.992233276367188, -31.018856048583984], [-28.332805633544922, -26.319400787353516, -27.32610321044922], [-28.332805633544922, -26.319400787353516, -27.32610321044922], [-25.740184783935547, -23.31174087524414, -24.525962829589844], [-23.746183395385742, -22.312318801879883, -23.029251098632812], [-22.12557601928711, -21.94270896911621, -22.034143447875977], [-20.299030303955078, -25.197311401367188, -22.748170852661133], [-23.971975326538086, -28.33308982849121, -26.15253257751465], [-29.992576599121094, -26.02522087097168, -28.008899688720703], [-32.645145416259766, -27.963478088378906, -30.304311752319336], [-30.462310791015625, -29.992691040039062, -30.227500915527344], [-29.992748260498047, -28.7193660736084, -29.356056213378906], [-26.319915771484375, -33.9842529296875, -30.152084350585938], [-26.319915771484375, -33.9842529296875, -30.152084350585938], [-27.609134674072266, -29.547332763671875, -28.57823371887207], [-30.462539672851562, -30.959012985229492, -30.71077728

Figure 73. Inference request data fetched from the monitoring system

11. Orchestration

All previous sections have demonstrated how to apply transformations and processing steps to data via a Deep Learning pipeline, in order to go from raw data into an ML model. So far all steps had to be manually executed in a proper order to produce the output model. However, once that process is established it can then be quite repetitive in nature. That means it decreases the efficiency of data scientists whose primary focus is on improving the accuracy of the models by applying novel techniques and algorithms. Such a repetitive process then should be automated and managed easily with the help of software tools.

One such tool is Apache Airflow (Airflow) [2], a platform to programmatically schedule and monitor workflows. Airflow is built on top of three core concepts: DAGs, Operators, and Tasks [10]. A Directed Acyclic Graph (DAG) is a model of the tasks you wish to run defined in Python. The model is organized in such a way that clearly represents the dependencies among the tasks.

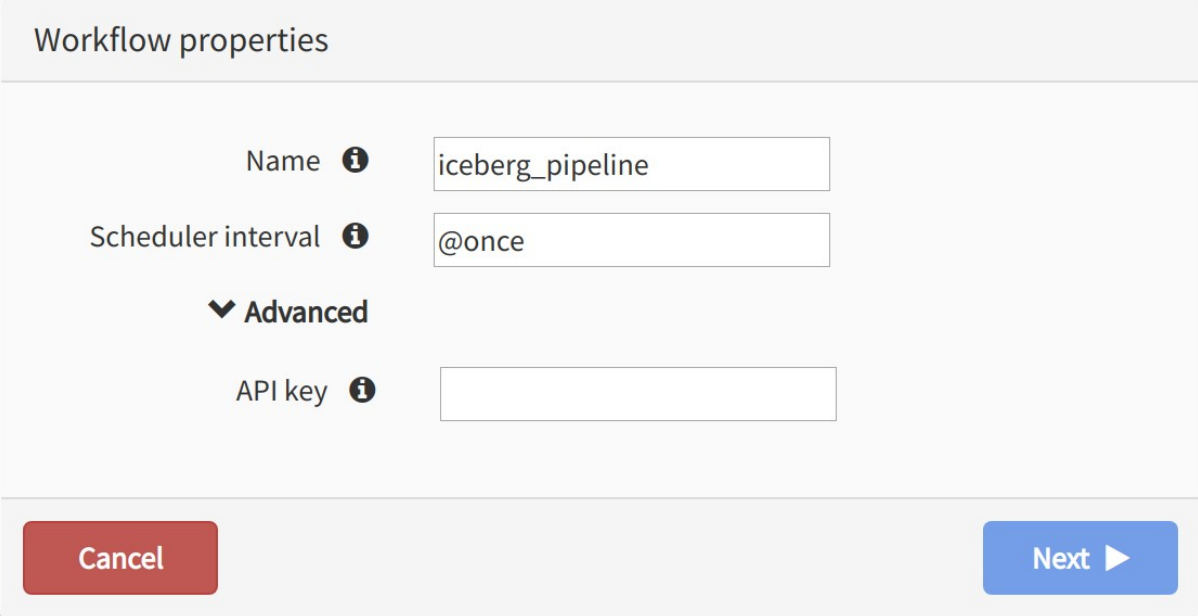
A DAG constructs a model of the workflow and the tasks that should run. So far a task is quite general, operators define what a task should actually execute. Operators are usually (it is recommended) atomic in the sense that they can stand on their own without sharing state with others. Tasks are instantiated operators at a specific point in time. Since workflows can be scheduled to run repeatedly, operators' results may vary. So tasks are also defined by the time which ran.

To this end, Hopsworks has extended Airflow in three ways:

1. Airflow has been integrated as a multi-tenant service accessible from a Hopsworks project. Members of a project are allowed to access only the Airflow workflows (DAGs) that are uploaded in the workflows directory of their project.
2. By providing Hopsworks specific Airflow operators that facilitate the development of creating workflows that implement DL pipelines. Development of Airflow workflows in Hopsworks typically involves launching Jupyter notebooks or PySpark programs.
3. By providing an intuitive UI as part of the Airflow service in a Hopsworks project, that enables users to define the order of task execution by selecting the jobs and the operators/sensors from.

Implementation of these operators and sensors is available in the logicalclocks GitHub repository [24] in the form of source code. The notebooks presented so far can be chained together in a series of tasks that can run in sequence or in parallel. Users define the order of task execution and how tasks depend on each other. This definition in vanilla Airflow is implemented by a Python program that users need to develop. The program uses Airflow Python libraries to create the DAG which can then be administered and monitored from the Airflow UI. A new UI, the Airflow DAG composer, has been developed in Hopsworks that enables users to create workflows by selecting Hopsworks Airflow Operators and Sensors. Users need to first create a job with the selected file to run being a notebook. The python

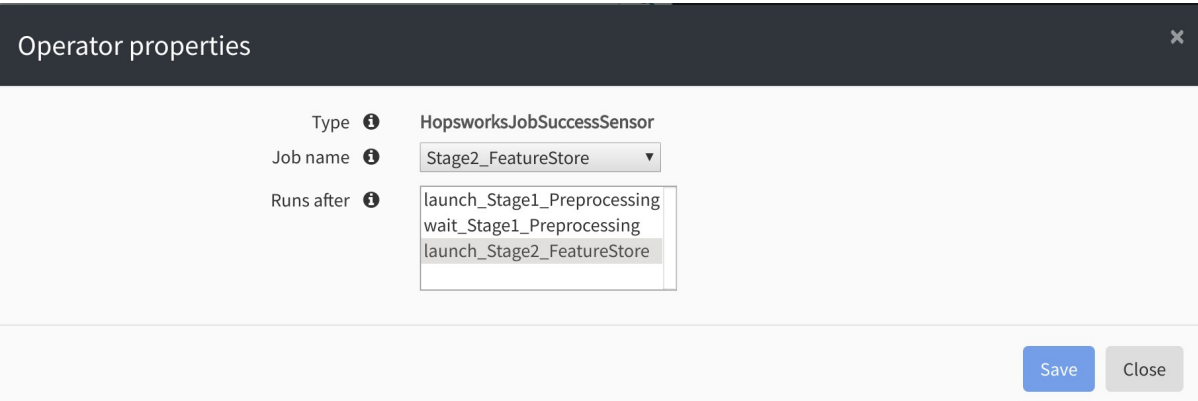
program `iceberg_pipeline.py` [23] defines the DAG and is auto-generated from the DAG composer. Pipelines can be scheduled and the first step of the DAG composer is to provide a name for the DAG and an optional schedule. If no schedule is defined, the DAG needs to be started manually from the Airflow UI. In addition, an optional parameter for an API key is provided for DAGs that need to be triggered from a remote instance and not from within Hopsworks. Figure 74 shows this first step of the DAG creation wizard.



The figure shows a 'Workflow properties' form. It has a title bar 'Workflow properties'. Below it, there are three input fields: 'Name' with the value 'iceberg_pipeline', 'Scheduler interval' with the value '@once', and 'API key' which is empty. There is an 'Advanced' section with a dropdown arrow. At the bottom, there are two buttons: 'Cancel' (red) and 'Next' (blue with a right arrow).

Figure 74. New Airflow workflow wizard

Figure 75 shows how users can easily set the HopsworksOperator properties, which are which job to run and after which task this job should run, hence creating a dependency in the graph (DAG).



The figure shows an 'Operator properties' form. It has a title bar 'Operator properties' with a close button (X). Below it, there are three input fields: 'Type' with the value 'HopsworksJobSuccessSensor', 'Job name' with the value 'Stage2_FeatureStore', and 'Runs after' with a list of tasks: 'launch_Stage1_Preprocessing', 'wait_Stage1_Preprocessing', and 'launch_Stage2_FeatureStore'. At the bottom, there are two buttons: 'Save' (blue) and 'Close' (grey).

Figure 75. New Airflow workflow wizard - Operators

Figure 76 shows the available operators and sensors and the ones that the user has selected. By clicking the “Generate Airflow DAG” button, the python file of the DAG is generated and persisted in Hopsworks.

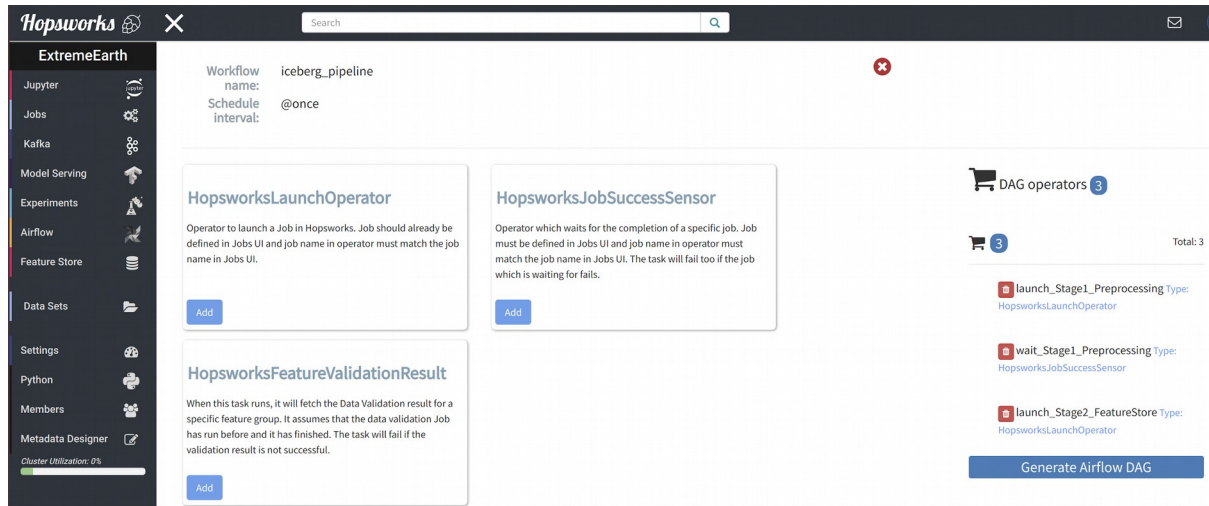


Figure 76. Airflow service UI in Hopsworks

Figure 77 shows the Tree View of the workflow from the Airflow UI. The latter provides a plethora of different types of views and tools to manage the lifecycle of a workflow.

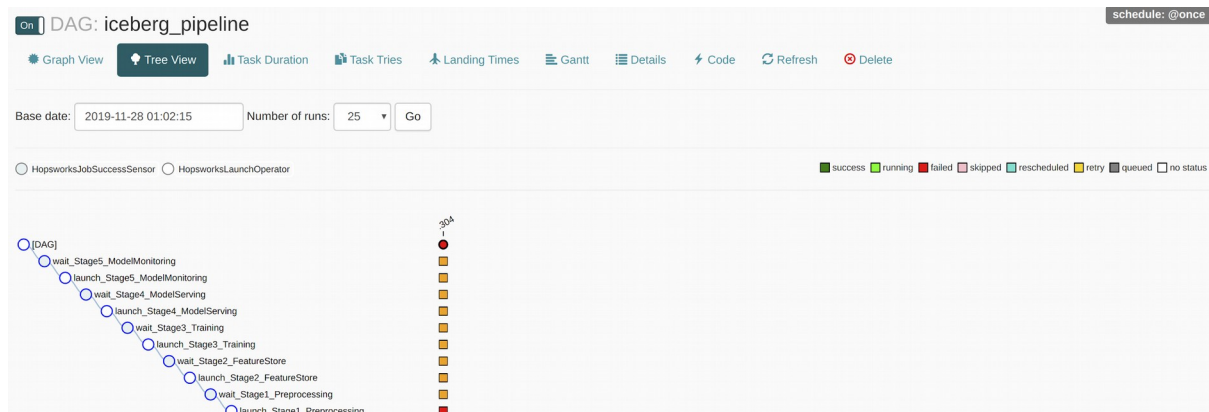


Figure 77. Airflow tree-view tasks

12. Hopsworks TEPs Integration

This section has been expanded and moved to deliverable “Platform software architecture - version II”.

13. References

- [1] "Say Hello to Asynchronous Search for PySpark"
<https://towardsdatascience.com/say-hello-to-asynchronous-search-for-pyspark-64c692a05595> [Online; accessed 1-June-2021]
- [2] "Apache Airflow" <https://airflow.apache.org/> [Online; accessed 15-November-2019]
- [3] "Creodias data access interfaces" <https://creodias.eu/data-access-interfaces>. [Online; accessed 2-April-2019]
- [4] "S3/Swift REST API Comparison Matrix"
https://docs.openstack.org/swift/latest/s3_compat.html [Online; accessed 22-November-2019]
- [5] "PythonResource.java"
<https://github.com/logicalclocks/hopsworks/blob/v1.0.0/hopsworks-api/src/main/java/io/hops/hopsworks/api/python/PythonResource.java> [Online; accessed 22-November-2019]
- [6] "Hopsworks REST API - Swagger"
<https://app.swaggerhub.com/apis-docs/logicalclocks/hopsworks-api/2.2.0> [Online; accessed 5-May-2021]
- [7] "Boto3 Documentation"
<https://boto3.amazonaws.com/v1/documentation/api/latest/index.html> [Online; accessed 22-November-2019]
- [8] "Spark Integration with Cloud Infrastructures"
<https://spark.apache.org/docs/2.4.3/cloud-integration.html> [Online; accessed 22-November-2019]
- [9] "Hopsworks UploadService.java"
<https://github.com/logicalclocks/hopsworks/blob/v1.0.0/hopsworks-api/src/main/java/io/hops/hopsworks/api/util/UploadService.java#L322> [Online; accessed 23-November-2019]
- [10] "Hopsworks read the docs Airflow" https://hopsworks.readthedocs.io/en/latest/user_guide/hopsworks/airflow.html. [Online; accessed 23-November-2019]
- [11] "SPIP: Accelerator-aware task scheduling for Spark"
<https://issues.apache.org/jira/browse/SPARK-24615> [Online; accessed 27-November-2019]
- [12] "Logical Clocks Spark GitHub repository"
<https://github.com/logicalclocks/spark/tree/branch-2.4> [Online; accessed 27-November-2019]
- [13] "Feature store"
https://hopsworks.readthedocs.io/en/1.0/user_guide/hopsworks/featurestore.html [Online; accessed 27-November-2019]
- [14] "Apache Hive" <https://hive.apache.org/> [Online; accessed 27-November-2019]
- [15] "MySQL Cluster" <https://www.mysql.com/products/cluster/> [Online; accessed 27-November-2019]
- [16] "Pandas" <https://pandas.pydata.org/> [Online; accessed 27-November-2019]
- [17] "pandas_helper.py"
https://github.com/logicalclocks/hops-util-py/blob/7ae489ce918732f3dd23608e8d268a3686664cab/hops/pandas_helper.py [Online; accessed 27-November-2019]
- [18] "End-to-end D1.8 demo pipeline"

- <https://github.com/ExtremeEarth-Project/eo-ml-examples/tree/main/D1.8> [Online; accessed 27-May-2021]
- [19] “Maggy” <https://github.com/logicalclocks/maggy> [Online; accessed 27-November-2019]
- [20] “Apache Kafka” <https://kafka.apache.org/> [Online; accessed 27-November-2019]
- [21] “Apache Avro” <https://avro.apache.org/docs/1.8.1/spec.html> [Online; accessed 27-November-2019]
- [22] “hops-util-py serving”
<https://github.com/logicalclocks/hops-util-py/blob/v1.0.0.0/hops/serving.py> [Online; accessed 27-November-2019]
- [23] “iceberg_pipeline.py”
https://github.com/ExtremeEarth-Project/eo-ml-examples/blob/main/D1.8/iceberg_pipeline.py [Online; accessed 27-November-2019]
- [24] “Hopsworks Airflow operators and sensors”
https://github.com/logicalclocks/airflow-chef/tree/1.0/files/default/hopsworks_plugin [Online; accessed 30-November-2019]
- [25] “Deequ” <https://github.com/aws-labs/deequ> [Online; accessed 30-November-2019]
- [26] Sheikholeslami, Sina, et al. “AutoAblation: Automated Parallel Ablation Studies for Deep Learning.” Proceedings of the 1st Workshop on Machine Learning and Systems. 2021.
- [27] “Model Training notebook”
https://github.com/ExtremeEarth-Project/eo-ml-examples/blob/main/D1.8/%5Bdone%5DStep3a_Model_Training.ipynb [Online; accessed 21-May-2021]
- [28] “Chef” <https://www.chef.io/> [Online; accessed 1-December-2019]
- [29] Apache Beam <https://beam.apache.org/> [Online; accessed 1-December-2019]
- [30] “Apache Flink” <https://flink.apache.org/> [Online; accessed 1-December-2019]
- [31] “Apache Beam Portability Framework”
<https://beam.apache.org/roadmap/portability/> [Online; accessed 1-December-2019]
- [32] “Python Streaming Pipelines on Flink - Beam Meetup at Lyft 2019”
<https://www.slideshare.net/ThomasWeise/python-streaming-pipelines-on-flink-beam-meetup-at-lyft-2019> [Online; accessed 1-December-2019]
- [33] Ismail, Mahmoud, et al. “ePipe: Near Real-Time Polyglot Persistence of HopsFS Metadata.” 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2019, Larnaca, Cyprus, May 14-May17, 2019. 2019.
- [34] “Iceberg demo distributed training”
https://github.com/ExtremeEarth-Project/eo-ml-examples/blob/main/D1.8/%5Bdone%5DStep3c_Model_Training_Distributed.ipynb [Online; accessed 21-May-2021]
- [35] “ELK stack” <https://www.elastic.co/what-is/elk-stack> [Online; accessed 1-December-2019]
- [36] “Karamel” <http://www.karamel.io/> [Online; accessed 8-December-2019]
- [37] “Statoil/C-CORE Iceberg Classifier Challenge”
<https://www.kaggle.com/c/statoil-iceberg-classifier-challenge/data> [Online; accessed 8-December-2019]
- [38] “CREODIAS” <https://creodias.eu/> [Online; accessed 10-December-2019]
- [39] “OpenStack” <https://www.openstack.org/> [Online; accessed 10-December-2019]

- [40] Meister, Moritz, et al. "Maggy: Scalable Asynchronous Parallel Hyperparameter Search." Proceedings of the 1st Workshop on Distributed Machine Learning. 2020.
- [41] Apache Spark, Logical Clocks fork.
<https://github.com/logicalclocks/spark> [Online; accessed 23-December-2019]
- [42] Prometheus metrics monitoring service.
<https://github.com/prometheus/prometheus> [Online; accessed 5-May-2021]
- [43] Grafana monitoring framework. <https://grafana.com/> [Online; accessed 23-December-2019]
- [44] Jupyter notebook. <https://jupyter.org> [Online; accessed 5-May-2021]
- [45] Anaconda Python distribution.
<https://www.anaconda.com/distribution/> [Online; accessed 23-December-2019]
- [46] Pandas. <https://pandas.pydata.org/> [Online; accessed 23-December-2019]
- [47] TensorFlow. <https://www.tensorflow.org/> [Online; accessed 23-December-2019]
- [48] Keras <https://keras.io/>. [Online; accessed 23-December-2019]
- [49] Chicago taxi rides dataset.
<https://data.cityofchicago.org/Transportation/Taxi-Trips/wrvz-psew> [Online; accessed 23-December-2019]
- [50] TensorFlow Extended (TFX). <https://www.tensorflow.org/tfx>
- [51] Scikit-learn. <https://scikit-learn.org/stable/> [Online; accessed 23-December-2019]
- [52] OpenStack. <https://www.openstack.org/> [Online; accessed 23-December-2019]
- [53] How we secure your data with Hopsworks
<https://www.logicalclocks.com/blog/how-we-secure-your-data-with-hopsworks> [Online; accessed 5-May-2021]
- [54] nbconvert: convert Notebooks to other formats
<https://nbconvert.readthedocs.io/en/latest/> [Online; accessed 5-May-2021]
- [55] Feature Store documentation
https://docs.hopsworks.ai/2.3.0-SNAPSHOT/generated/feature_validation/ [Online; accessed 6-May-2021]
- [56] ESA SNAP Toolbox <https://step.esa.int/main/toolboxes/snap/> [Online; accessed 6-May-2021]
- [57] Kubernetes jobs <https://kubernetes.io/docs/concepts/workloads/controllers/job/> [Online; accessed 6-May-2021]
- [58] esa-snap docker image <https://hub.docker.com/r/atavares/esa-snap> [Online; accessed 7-May-2021]
- [59] GDAL <https://gdal.org/> [Online; accessed 7-May-2021]
- [60] GDAL Python package <https://pypi.org/project/GDAL/> [Online; accessed 7-May-2021]
- [61] Hopsworks Feature Store tags
<https://docs.hopsworks.ai/latest/generated/tags/#tag-schemas> [Online; accessed 11-May-2021]