University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Krzysztof Antoniak
Student no. 358985

Michał Jadwiszczak
Student no. 406163

Jan Kukowski
Student no. 406204

Maciej Procyk
Student no. 406304

Integrating Nussknacker with selected Machine Learning tools

Bachelor's thesis
in COMPUTER SCIENCE

Supervisor:
mgr Grzegorz Grudziński
Institute of Informatics

Warsaw, June 2021

Abstract

In the thesis, we present a new implementation of Nussknacker's enricher named Prinz that is responsible for Machine Learning models management and scoring. We describe the source of the need for such implementation and the background behind the Event Stream Processing approach. We show our approach to the process of developing the open-source library with API specification and our API implementations including MLflow models registry, PMML standard, and H2O models registry. We describe the process towards managing open source projects, developers' environment management for the process of development, and the decisions made to create our implementation of Nussknacker's enricher. Additionally, we present our thoughts on the topic of each integration, its development process, and the final results of API implementation which includes additional assumptions made during the deployment of integration. As a final result of our work with the project, we provide working library sources with readable examples of its deployment to the Nussknacker environment.

Keywords

Prinz, Nussknacker, Scala, Machine Learning, models scoring, ML integration, MLflow, PMML, JPMML, H2O, H2Oai, Apache Flink, Event Stream Processing, ESP, ML models registry

Thesis domain (Socrates-Erasmus subject area codes)

11.3 Informatics, Computer Science

Subject classification

Software and its engineering – Search-based software engineering

Tytuł pracy w języku polskim

Integracja projektu Nussknacker z wybranymi narzędziami Machine Learning

Contents

Chapter 1

Introduction to Nussknacker and Prinz

1.1. Nussknacker

Nussknacker is an event-stream processing and decision-making solution developed by TouK, a software house based in Warsaw. It's a fully open-source project available on GitHub. The project has been in development since late 2016, and at the moment of writing this thesis it has almost 200 stars, 35 contributors, and a codebase of almost 180 thousand lines. Nussknacker lets the user design, deploy, and monitor streaming processes through an easy-to-use GUI. It's intended as a simple way for non-programmers to write and customize processes. The user creates a diagram describing the flow of data from many types of blocks available in the Nussknacker UI (e.g. filters or aggregators). The described process can then be tested and run on an Apache Flink cluster.

1.1.1. Use cases

Historically, the initial use case for Nussknacker was Real-Time Marketing or RTM. One of TouK's clients had some large data streams, which they intended to use for their marketing campaigns. The key here was the ability to process and manipulate the data quickly. However, most modern stream processing engines require the user to know a domain-specific programming language. Therefore, Nussknacker allowed non-technical users - like analysts or managers to process large amounts of data and draw actionable insights.

Nowadays, the other main use case for Nussknacker is fraud detection, in particular in the telecom business. When dealing with some kinds of fraud (e.g. SMS spamming) it's necessary to take instant, automated action. Changes to those actions shouldn't each time require additional development. It's especially important for companies that might not have an internal programming team. In such a case, analysts and other users with little or no programming background can design and monitor the processes themselves using Nussknacker.

1.2. Prinz

Prinz is a library of extensions for Nussknacker. It provides a simple API, that allows developers to add new integrations with machine learning engines or repositories. At the moment integrations with 3 tools - MLFlow, PMML, and H2O are available in Prinz. Prinz integrations are highly configurable - each can provide its way of storing the models and retrieving data from them. Each integration includes one or more model repositories that are used for different model accessing strategies. When integration is added to Nussknacker, each model listed in those repositories becomes available in the Nussknacker UI as a block. This block can then be integrated into the normal flow of the designed process and for example used as a filter or an aggregator.

Chapter 2

Basic terms

2.1. Event streaming processing

Nussknacer is an example of an event streaming process application. ESP is taking some actions on a series of data that comes from a system that creates data continuously. That action can be a typical data processing functions like aggregations (e.g. summing), analytics (e.g. making predictions), transformations (e.g. parsing data), enrichment (e.g. combining data) and ingestion (e.g. inserting).

This kind of data processing method is required when we need to take some steps as soon as possible or even predict incoming events. Thus, event stream processing is often called real-time processing.

To implement such an application, in general, we need to take care of two issues to process a continuous stream of data. The first thing is to store incoming data and then process this data using some program. One of the most popular technologies to develop ESP is Apache Kafka. We will describe it in the following paragraphs.

The event streaming process is widely used in the financial business to process payments, real-time marketing, detect anomalies and frauds. Also, the Internet of Things takes use of ESP to for instance identify radio-frequency identification. In most production applications, developers use it to monitor processes. All of these examples rely on continuously delivered data.[9]

2.2. Apache Flink

As mentioned above, Apache Flink is one of the most popular events streaming processing technology. Flink is an open-source project by Apache, that is capable of stream and batch processing. Nussknacker originally started as Flink "self-service" process authoring tool and it is still utilized by it as the main processing engine.

In Apache Flink, continuous streams are also called unbounded streams and batches - bounded streams. While unbounded streams have only defined the beginning of it but no ending, the bounded ones have both start and end. To process continuous streams, we often require them to be in a specific order, such as the timestamp of the event. On the other hand, if we are processing batches, we have a full set of data, so we can sort it on our own.

Apache Flink is built in such a way to be easily run at any scale. Applications are parallelized into tasks, which are distributed and concurrently executed in a cluster. Flink is capable of maintaining a very large application state by utilizing its asynchronous and incremental checkpointing algorithm. Use of them ensures minimal impact on processing latencies, however, it guarantees exactly once state consistency.[1]

2.3. Fraud detection

Over the last decade's increasing amount of financial transactions via the Internet, creates more opportunities for criminals and daily users to try to hack the shopping systems. The value of the global E-Commerce market is estimated to reach $4.9 trillion this year. According to the Internet Crime Complaint Center, internet crimes in the year 2019 were the highest ever by then. People and businesses lost almost $3.5 billion more than in 2018. We can categorize internet fraud into some categories like payment fraud, identity theft, document forgery, and account takeover.

Machine learning can perfectly address this issue. Due to the enormous quantity of financial transactions, data analysts have enough data to train ML models and teach them the pattern of internet frauds. Machine learning is generally faster and more efficient than traditional algorithms. Because E-Commerce is such a dynamic environment, data scientists would have to constantly adjust traditional algorithms, unlike machine learning models' ability to easily and self-adapt. Models also take over repetitive and monotonic work of detecting the frauds manually. Also, machine learning is more scalable, meaning - the more samples we fit the model with, the better it gets.[4]

Chapter 3

Architecture overview

This chapter describes Prinz's approach to integrating different tools with Nussknacker. We will discuss our intermediate model representation in a typical process deployment.

Prinz extensions are injected into a Nussknacker instance in the configuration phase. Instance administrator should configure model repositories that discover and provide model instances. Each model is interpreted and exposed to Nussknacker UI as an enricher. When the Nussknacker process is deployed, Prinz translates parameters for the external model, delegates scoring, and brings results back to the process.

3.1. Model discovery

Nussknacker configuration should use one or more instances of `ModelRepository`. Repositories handle listing models available in configured storage. Successful exchange with storage results in a list of model handles, which will be converted to enrichers.

Prinz comes with ready-to-use implementations using local filesystem or HTTP endpoint. The HTTP server is expected to expose a list of models as JSON. Developers can provide custom implementations of `ModelRepository`.

3.2. Models

Prinz uses an intermediate internal model representation for unified handling of integrations. `Model` trait represents a single entry in the Nussknacker UI. Model is identified by name and may contain version info. For some integrations, it is possible to fetch metadata without loading the whole model.

By calling `Model.toModelInstance` system instantiates the model. Exact behavior is implementation dependent, but this call should:

- prepare model for signature extraction,
- prepare model for scoring.

The operation may require additional communication with the service. At this point, model would usually get loaded into memory if it is needed.

3.2.1. Signatures

Each machine learning model has inputs and outputs. Their names (or ordering) along with data types form a model signature. Since the range of supported types varies between different libraries, Prinz also abstracts signatures.

Each `ModelInstance` is instantiated with a signature provider, which will supply inputs and outputs of the model. External data types are mapped to internal `SignatureType`s based on types supported by Nussknacker. Extracted features and outputs are available in the Nussknacker UI for process designers.

3.3. Scoring

In a deployed Nussknacker process events move through the flow sequentially triggering computations at each step. When computation reaches a Prinz enricher, it triggers scoring for supplied inputs. (At this point system does not support batching). Scoring starts by calling `run` on a `PrinzEnricher`. During the scoring phase Prinz:

1. converts inputs to external types based on the model signature,
2. triggers scoring and receives outputs,
3. translates results from external types to the ones used by Nussknacker.

Received outputs may be used for further processing.

Chapter 4

Prinz API

This chapter describes the API provided by Prinz. The main goal of the API is to allow developers to add new integrations in a simple and unified way. Each consecutive integration consists of customizable classes that extend traits defined in the API. Traits that have to be implemented in every integration include `SignatureProvider`, `Model`, `ModelInstance`, and `ModelRepository`. Moreover, the optional `ApiIntegrationSpec` trait includes a convenient way of testing the implemented integration and its use is highly recommended.

4.1. ModelSignature

The `ModelSignature` trait describes the correct format of input and output data for a given model. Each signature consists of two lists of `SignatureFields` - one for the input and one for the output. A `SignatureField` describes the format of a single column of data. It consists of a `SignatureName` - the name of the given column and a `SignatureType` - the type of data in this column, which can be interpreted by Nussknacker.

4.1.1. SignatureProvider

The `SignatureProvider` trait was created to separate the logic of extracting the signature from a given machine learning model. This separation makes the process of accessing the signature customizable for each integration. The trait consists of a single function `SignatureProvider.provideSignature`, which takes one argument: an object representing the location metadata of the signature. If a signature can be provided using this metadata, the function returns a `ModelSignature`. In some cases, the integrated tools may use a different typing system than the one available in Nussknacker. Therefore, the role of a `SignatureProvider` is also to translate all types present in the signature to Nussknacker-based ones.

4.2. Model and ModelInstance

The `Model` trait represents a top-level description of the machine learning model. It allows access to all necessary information about the machine learning model but is not runnable by itself. Calling the `Model.toModelInstance` function returns an instance of this model, which can later be scored using provided data. Moreover, through `Model.getMetadata`, the trait allows access to additional information about the model. This information includes the model version, name, and signature.

The `ModelInstance` trait represents a runnable instance of the machine learning model. The scoring proccess is started by calling the `ModelInstance.run` function. This function takes one argument: a multimap containing input data for the model instance. The keys of this multimap refer to the column names of the model input signature. During the scoring process, the input data is first verified against the model input signature. The `ModelInstance.verify` function is predefined, and it checks whether the column names in input data exactly match those present in the signature. If the input is correctly verifed, the `ModelInstance.runVerified` function is called. This function directly scores each row of data and is entirely implementation-dependent. If the whole process finishes without errors, a multimap containing output data is returned. The keys of the returned multimap refer to the column names of the model output signature.

4.3. ModelRepository

The `ModelRepository` trait provides a consistent way of listing models for each integration. Each instance of a `ModelRepository` contains information about the configuration of a given integration. Based on this information, the `ModelRepository.listModels` function returns a list of all `Model`s available in this repository. Those models can be accessed in the Nussknacker UI and scored using provided data after compiling the Nussknacker process.

Chapter 5

MLflow integration

5.1. MLflow project overview

MLflow is an open-source project which allows managing the whole machine learning lifecycle. It includes experimentation, deployment, and reproducibility of models as well as a central registry for versioning trained models. It was proposed by the Databricks team working with hundreds of companies using machine learning as a solution to common production problems when working with machine learning models and their deployment.[6]

The whole MLflow platform was inspired by existing solutions from Facebook, Google, and Uber which seem to be limited because they support only a small set of built-in algorithms or a single library, and they are strictly connected to each company's infrastructure. They don't allow easy use of new machine learning libraries and share the results of work with an open community.[11]

MLflow was designed to be an open solution in two senses:

- open source: it is an open-source library publicly available that can be easily adapted by users and extended to the expected form. Additionally, the MLflow format makes it easy to share the whole workflow and models from one organization to another if you wish to share your code with collaborators.
- open interface: it is designed to work with many already available tools, machine learning libraries, implemented libraries, and algorithms. It's built using the REST APIs and readable data formats where for example a machine learning model can be seen as a single lambda function called on model evaluation. It was designed to be easily introduced to existing machine learning projects so the users can benefit from it immediately.

5.2. Trained models management

MLflow allows users to use existing models and easily convert them to open interface format. When working with the MLflow framework we used the Python language for creating some Machine Learning models for test purposes using the sklearn framework which is one of the many frameworks supported by MLflow. Then the pre-trained models were imported to the MLflow repository which saved their state in an external database and some data storage server that was responsible for keeping the artifacts from training. In practice, the artifacts' data that was useful in our case was the signature of the model which was exported to YAML file format.

The training process of used models was short enough that we can train the models every time the MLflow server was created in a clean environment. This approach to creating this environment allowed us to easily debug any inconsistency in models' training process because we could see the difference between the clean environment and the environment with trained models.

5.3. Trained models serving

MLflow allows to deploy the trained models from its registry in a few ways including:

- deploying `python_function` model to Microsoft Azure ML
- deploying `python_function` model to Amazon SageMaker
- exporting `python_function` as an ApacheSpark UDF
- exporting a REST API endpoint serving the model as a docker image
- deploying the model as a REST API local server

Using any of these approaches may be easier or harder to deploy in practice depending on the architecture that we already work with and the resources that can be used for the deployment process. Managing the local The REST API server is the easiest solution while it doesn't scale up with the number of models. When the developers take care of the environment, and it's really hard to set up a fresh, clean environment from scratch it's better to manage the models using separated Docker containers (which deployment process can additionally be easily automated).[7]

In our environment, the models after training are served as a local REST API in Docker image with MLflow the server inside where they have ports exposed for external usage. With this approach, we can clean the environment on every setup of integration tests of the library, download fresh, prepared images, and train the prepared models, which are then stored in MLflow clean database and S3 storage (that are prepared as separate Docker images).

5.4. MLflow Repository

Configured MLflow server used in our environment model serves as the model registry which takes care of model versioning and serving the base models' information using the REST API. As there is no implementation for Scala (and Java) language for this REST API, we have created our implementation of the provided API with the usage of a high-level HTTP client library named `sttp` written by SoftwareMill and the model objects serializer called `circe` powered by Cats. These two libraries allow us to create the model corresponding to the official MLflow documentation which could be easily integrated with the current version of the MLflow models repository.

The whole model of data served by the MLflow models' registry server is provided in JSON format, for which we managed to create corresponding models in code with the usage of compile-time code analysis with the Scala macros sbt plugin. In this approach, it was enough to create the case classes with proper fields to get the automatic conversion between the received text data in JSON format and the objects model in the programming language. The whole process is based on JsonCondec code annotation which allows to preprocess case classes in Scala in compile-time and generate the Encoder and Decoder objects implementations for the annotated class. In the standard approach, the developer would have to manually write this code which would just translate the JSON fields to object's fields and vice versa while this can be in most cases done automatically.

5.5. MLflow Model and ModelInstance

MLflow model is instantiated based on the data received and parsed in JSON format from MLflow models registry and the signature data located in artifact storage of model. The signature data is parsed separately from a YAML formatted file which is also mapped to `JsonCodec` model class with auto Encoder/Decoder mechanism.

When the MLflow model is instantiated to allow scoring the served model, the whole conversion of data sent to the external model service is done by the MLflow model instance during the run method invocation. We created the abstract of Dataframe which corresponds to the single input for the model. When typed input data comes from another Nussknacker service to the Prinz MLflow implementation, it has to be manually converted to the valid JSON format, which can be determined based on the signature of the model. In the runtime of the process, the library receives the data in a unified format of type Any which then has to be recognized and parsed before creating the input frame for the model. MLflow served models support two types of JSON data providing named split and record orientations for compatibility and easier work with different types of input data. In our model we use only a single approach of Dataframe to JSON conversion as the data received from another service is always given in the same format. The data conversion process which is needed before sending the data to the model and after receiving the data from the model was implemented in the separated object's MLFDataConverter methods as they don't depend on the model and exist in the model architecture as the static methods which are responsible only for data conversion for this single type of implementation.

5.6. MLflow Model Signature

MLflow model signature is not a necessary part of a model registered in the models' registry provided by the MLflow library. In our approach to the problem, we assume that every user of our library should save the models in the registry in a way that includes steps in creating the model signature. We found this way of typing the model input and output as the easiest one because the final user of Nussknacker doesn't have to know anything about model implementation (when it was logged to the registry with a signature). Additionally creating the model with a typed signature ensures the user that only the valid input will be allowed for model input (which is not so obvious while most of the model management on the MLflow side is done with Python that hasn't the static typing). However, creating the signature for MLflow models in Python has a single drawback e.i. the model output doesn't have any names and is given as the ordered list in the JSON response. We manually create the additional labels for model outputs in the process of scoring with the following outputs named `output_0`, `output_1` etc. Thanks to this approach the final user of the Nussknacker GUI can configure the model more easily but still, he has to know the mapping between the interesting data and its order in the output model map.

It's worth noting that the saved model signature in the case of MLflow models is stored on external data services like Amazon S3 data storage which seems to be the default choice of users of the MLflow library. In Prinz, we implemented accessing the models' signatures located only on S3 storage and here we see the place for future improvements to provide some more generic approach of fetching the signature data.

Chapter 6

PMML integration

6.1. PMML project overview

Predictive Model Markup Language is a standard to represent predictive solutions. It was developed by Dr. Robert Lee Grossman and the Data Mining Group. The main goal was to be able to exchange predictive models produced by data mining and machine learning algorithms between applications. The standard is widely spread across the world, with over 30 organizations have announced their use of it.[2][3]

In our Prinz solution, we are using `JPMML` library to serve PMML models. `JPMML` is open-source Java's implementation of the PMML standard. This way we can support most of the standard versions, especially the latest 4.0, 4.1, 4.2, 4.3, and 4.4. Modules of the project used in Prinz are `JPMML-Model`, `JPMML-Evaluator` and `JPMML-Transpiler`. The first one is released under the terms and conditions of the BSD 3-Clause "New" or "Revised" License. Model and transpiler are published under GNU Affero General Public License v3.0. Therefore, that is the point of such license in PMML's package in our repo.[5]

6.2. PMMML model representation

PMML is based on XML language. The file is composed of several components such as:

- header - contains general information about the document,
- data dictionary - contains definitions for fields used by the model,
- data transformations - maps user data into the form used by the mining model,
- model - contains the definition of the data mining model,
- mining schema - list of all fields used in the model. Can be a subset of fields defined in the data dictionary,
- targets - allows for post-processing of the predicted value. Can also be used for classification tasks,
- output - allows naming desired output fields expected from the model.

6.3. PMML Repository

The standard as well as its `JMPPL` implementation, does not provide any systematic way of versioning and serving models. Thus, we had to design our way of storing PMML models. One of the project's key principles was to make Prinz open for further integrations and development. To ensure this we have implemented a `RepositoryClient`. A generic API `AbstractRepositoryClient` provides methods to list files, open them and validate. Then the `RepositoryClientFactory` knows which implementation of the client to choose based on URI's schema. For now, we have supported the local filesystem client and HTTP client. Working with a local file system repository is fairly straightforward. To get the HTTP repository to work, an HTML selector to a files' href elements is required. Then if URI leads to HTML page, Prinz will download the page find the links to PMML files. Otherwise, if URI points to the PMML file, the repository will simply get it.

To integrate a new one, you have to simply implement the abstract API and add one line to the factory's registry.

As mentioned previously, `PMMLModelRepository` is based on `RepositoryClient`. By our convention, the file name should be composed with model name and model version, separated by separator defined in the application's config. For instance, one of the PMML file's name in our GitHub repository is `PMML-FraudDetection-3-v0-1.pmml`, so it represents `PMML-FraudDetection-3` model in version `0-1`.

6.4. PMML Model and ModelInstance

`PMMLModel`s are created based on the data received from scanning the path of the repository. `RepositoryClient` returns payloads, where each of them contains basic information of models and input stream's source of a file. Then the actual `PMMLRepository` creates models' objects from that data.

`PMMLModel` holds the model's name and its version. It is also responsible for providing a signature. The process of providing the signature will be explained in short below.

`PMMLModelInstance` is using to create `PrinzEnricher`, which represents a tile on Nussknacker's flow. It is capable of actually running the model and mapping its output to be understood by Nussknacker. We can achieve this, using an evaluator from `JPMML-Evaluator` library and passing the enricher's input to it. The evaluator then takes care of actually running the model and returns the result.

6.5. PMML Model Signature

The PMML standard does not provide any separate method of retrieving the signature, rather than reading it from the file. To get the input signature and also output signature we are using the previously mentioned evaluator from `JPMML-Evaluator` library. Then `PMMLSignatureProvider` is converting each field of signature to the definition of fields properly typed in Prinz API and parsing types to types supported by Nussknacker. We are supporting most of the XMLSchema-2 types.

Chapter 7

H2O integration

7.1. Introduction

H2O is one of the leading machine learning platforms. It features an open-source server with in-memory implementations of multiple popular algorithms. Users can access the server through hosted visual notebooks or using multiple programming languages, including R and Python. AutoML functionality helps to automatically select matching model types for provided datasets. Created models can be deployed as POJOs[1] or MOJOs[2] for scoring.

H2O.ai, the company behind H2O, created the platform to democratize access to AI solutions. Its multiple products are fully open source, while the company provides paid enterprise support. With over 12000 customers in different industries, H2O.ai was classified as a Visionary in Gartner 2021 Quadrant for Data Science and Machine Learning, surpassing companies like Google and Microsoft in terms of *Completeness of vision.*

7.2. H2O in Prinz environment

H2O provides both a server endpoint and a set of Java libraries for automated scoring based on deployment files. We chose to load files directly and score them in memory by libraries, mainly because of an open license (Apache 2.0.) and compatibility between Java and Scala languages. To score incoming data, our implementation performs the following steps:

1. load model, from memory, file or URL,
2. map Nussknacker types to H2O types (`String` or `Double`),
3. convert incoming data to rows,
4. score rows individually,
5. extract output values and retype outputs for Nussknacker,
6. gather results.

This approach creates a few challenges, described in the following sections.

[1]POJO – *Plain Old Java Object*

[2]MOJO – *Model Object, Optimized* – is an alternative storage standard by H2O.ai.

7.3. Loading models

When creating a model, H2O outputs POJO or MOJO files which can be downloaded by the user. The scoring library needs to know the file type and how to acquire its data. Since Nussknacker UI displays all exposed models, it needs to know about them when creating enrichers in the configuration phase.

We chose MOJO as a supported type. POJO seems to be a predecessor format with limitations on file size and worse performance. While the company will support POJO files, MOJO is the recommended file format [8].

As for locating data, there is no standard for discovering models. One approach could be to download models directly from the H2O server, but this defeats the purpose of using small, portable files. This also prevents infrastructure setups without an exposed server.

For simplicity and coherence, we opted for a custom server implementation based on the same interface as in PMML integration. Any repository client implements methods for listing and accessing files. Built-in implementations allow for using files from a local directory or loading a list of URLs from a remote endpoint. For details on PMML implementation, see section 6.3.

7.4. Extracting signatures

General properties of the model (like name or version) can be provided by the repository as in the case of the PMML repository. All metadata about the model contents is described inside deployment files. Because of this, we have to load all file models when loading configuration, even if a particular model won't be used. On the flip side, H2O libraries have full knowledge of the model and can provide names and types of inputs and outputs without any further analysis.

7.5. Scoring models

At this point, Prinz does not support entry batching, but internal API operates on sets of rows. H2O libraries can score only single rows, so sets have to be ungrouped into single rows, scored, and then grouped back. The input data for scoring models has to be converted before evaluating the H2O model instance. This step is needed as in the whole flow of data inside Nussknacker, the String values don't have extra quotes while the H2O needs to have the non-numerical input data enclosed in extra quotes.

The result of scoring is a *prediction*, with exact representation depending on type of the model. H2O supports some other types of algorithms beyond traditional ML models. Predictions have to been transformed to extract output values for Nussknacker. Currently supported types are:

- `AnomalyDetection`,
- `Binomial`,
- `Clustering`,
- `CoxPH`,
- `KLime`,
- `Multinomial`,
- `Ordinal`,
- `Regression`.

Chapter 8

Development environment

8.1. Development environment overview

We created a specially configured development environment as a part of our project repository to simplify the process of the environment setup which is needed to run every integration with our library and Nussknacker service. For this purpose a separate directory named `dev-environment` was created which includes all the Docker configurations and scripts for setting up the environment from scratch.

The base for configuration for the environment is made of the .env file which contains the environment constant variables' definitions and allows easily changing the ports of services and some paths in Docker images that can be configured. This file is loaded also by the sbt which is then able to work with the same environment for integration tests that use the Docker services to test every integration.

The process of creating the new environment comes to running a single bash script from the `dev-environment` which allows configuring the build process with additional flags specified for script execution. There is a possibility to run only a single integration environment (e.g. only MLflow server repository with its environment) or to exclude the process of library recompilation before placing it in the Nussknacker image. All these improvements were added during the process of adding new integrations as we found it annoying to spend a lot of time waiting for some part of the environment that wasn't needed to test and experiment with another integration. Additionally, we created the extra bash script for cleaning the environment cached data as discovered to take a lot of disc space when having so many different integrations with their environments in a single project.

Every integration has its `docker-compose` configuration file which specifies the Docker services needed to run the integration environment. Additionally, there is a separate configuration named env for Nussknacker image with its services, where the compiled Prinz library is placed. Thanks to such an approach it is easier to set up every environment separately and run only the needed ones. However, this also makes it a little harder to allow communication between every environment - we need to manually add the Docker network to which the setup environments are connected during creation time and then can communicate easily. Moreover, we decided not to expose every integration port for external services for the other integrations and Nussknacker environment but create a single proxy server for each integration which works as a barrier between the integration implementation details and the outside world (and then the environment works more like in real life scenario).

The Docker images for each integration needs extra dependencies which are managed using the `conda` environment manager. There is quite a lot of work to be done by installing all of them separately so we decided to create the Docker images for integration and publish them in the external Docker images repository. We decided to use the GitHub images hub which allows us to publish the images as a part of our open-source project but unfortunately forces the user to log to GitHub before downloading the image. However, this GitHub policy can change shortly as the community doesn't seem to like the logging requirement and there are many open discussions on this topic.

8.2. Models serving in environment

Every integration includes individual ways of models creating and serving them after the training process:

- MLflow models are trained after creating the models' registry environment and then served as Docker services from the same Docker image. Their specification is saved on a separate PostgreSQL database and the signatures are kept on the S3 storage provided by the Minio Docker image. This setup makes the prepared environment behave like a production version of the MLflow server because there is no usage of local storage for data-keeping.

- PMML models are exported as the XML files during training on image start and served as the files by the HTTP little server written in Python. This approach uses the PMML Python dependencies only during training time as there is no alternative for models management server in case of PMML models representation. The models in our approach have to be listed in a specific way to give the developers ability to automatically find their locations with the proper selector path which specifies the models' refs on the main site of the server and is configurable in Prinz.

- H2O server is somehow between the MLflow and the PMML integration as it sets up a full models registry server during training time and can list models and serve them with the usage of REST API. However, in our case, the simpler approach to H2O models was chosen and after training the models they are saved as the standard MOJO files and then led by the Prinz library as the local scoring models. Here we leave some room for future integrations as there is also a possibility to score the H2O models on the side of the H2O server like in the case of the MLflow registry.

8.3. Proxying models environments

For each integration, we created a separate proxy server based on light nginx alpine Docker image which is set up with only a single configuration file having needed specifications. In the case of MLflow integration, we tried to simulate the real-world scenario usage of the MLflow registry by manually setting the proxy configuration connected with the buffering of the data and setting some custom requests' headers. In the other integration, the served models as the XML and MOJO files are only proxied with the change of the port on which they are available to the user.

Furthermore, for each integration, the proxy also serves a few static files which contain the data needed in the phase of integration tests. It is responsible for serving like some simple REST API server that is capable of providing extra inputs for models in specific phases of library tests.

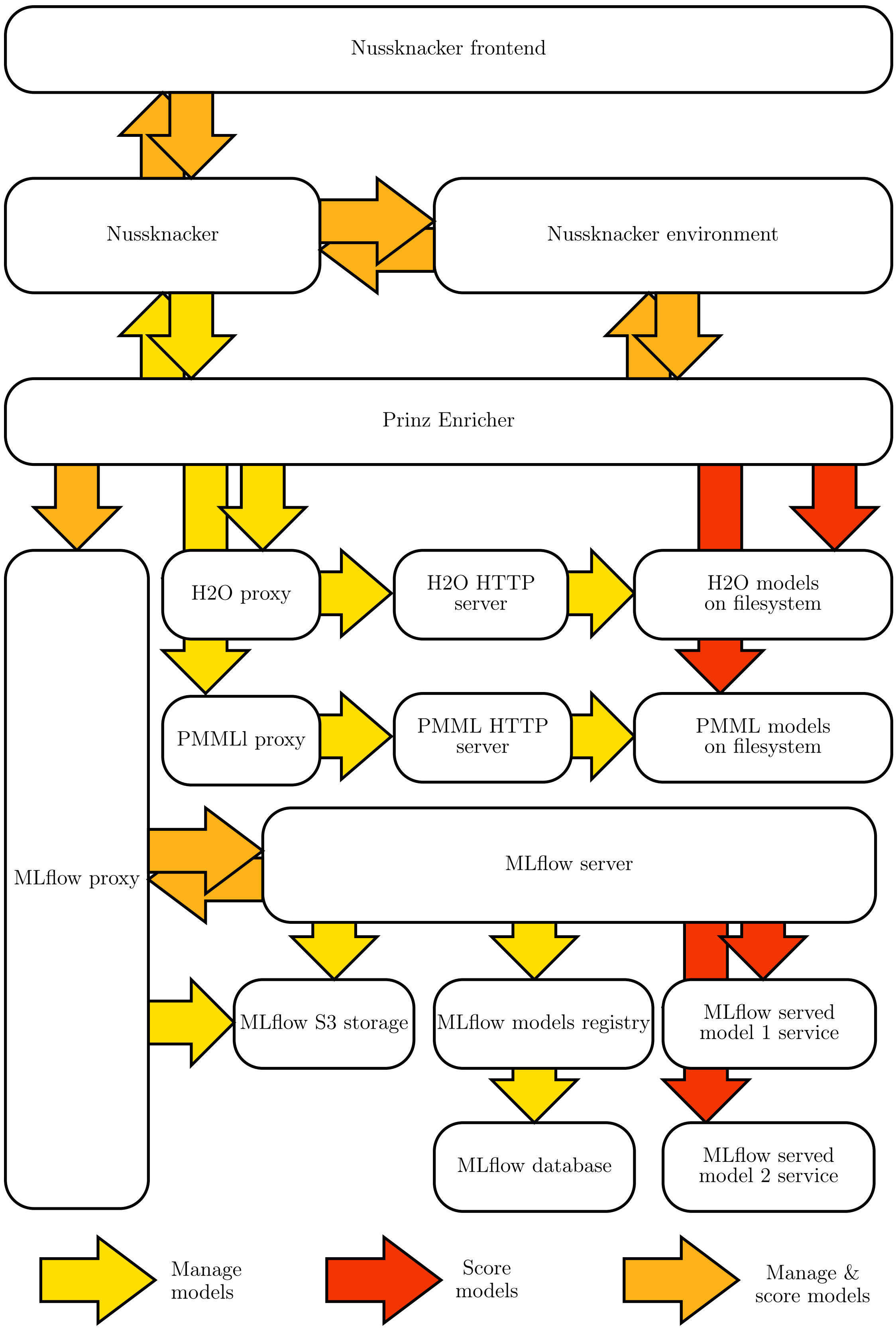
8.4. Development environment in integration tests

Each integration runs separately the integration tests which was one of the main reasons to separate the environment configurations into a few files. During the tests phase there is a need to know the project root file in the filesystem to locate these configuration files so before running any integration test the user has to define the `REPOSITORY_ABSOLUTE_ROOT` environment variable. After this initial configuration running tests from the console is as simple as calling a single sbt command as the whole environment configuration is loaded with the sbt plugin from the `.env` file. However, in the case of the need to run tests from an IDE, we need to specify separately the `.env` file in the test run configuration as for now there is no possibility to load this information automatically in the IDE.

The testing phase includes testing the integration part e.i. scoring the models available in integration models source and checking their signatures and other parameters. Moreover, there are tests which are checking the models ability to being proxied with some external data source (outside of Nussknacker environment source of data) so they use the described feature of nginx serving as a simple REST API and use the local H2 database as the source of data for tests run. All of these described tests are wrapped in the abstract traits specifications to allow running them on every integration with the minimal configuration process. The process of running the tests for specific integration includes setting up the environment from scratch but everything is done by the testcontainers library which uses the docker-compose YAML files. This approach connects the source code of our library and its tests with the environment definition while in the test running phase there is no need for the developer to manually set up the environment. Additionally, the unit tests which don't use the integrations' environment just run scala tests code without touching anything from the environment definition so there is an easy way to run them and verify some specified parts of code without the process of working with any Docker containers.

8.5. Development environment scheme

The overview scheme of described development environment is presented below. It shows the scheme of working with models via Nussknacker's frontend down to models listing and scoring models' instances.


Nussknacker frontend
Nussknacker
Nussknacker environment
Prinz Enricher
H2O proxy
H2O HTTP server
H2O models on filesystem
PMMLl proxy
PMML HTTP server
PMML models on filesystem
MLflow proxy
MLflow server
MLflow S3 storage
MLflow models registry
MLflow served model 1 service
MLflow database
MLflow served model 2 service
Manage models
Score models
Manage & score models

Chapter 9

Project development

This chapter describes approaches to managing work and developing the project.

9.1. Project management

9.1.1. Stakeholders

There were three parties involved in the project.

Development team. The authors of this thesis worked on the project for its whole duration. Students had little or no prior knowledge of Scala or machine learning solutions. Maciej Procyk took the role of a team leader on a day-to-day basis.

Touk sp. z.o.o. s.k.a. Touk[1] is a software house that ordered the integrations in response to rising demand for machine learning solutions in Nussknacker. Maciej Próchniak was the contact person and a technical consultant for the project.

University of Warsaw. The authors developed the project for a bachelor thesis at The Faculty of Mathematics, Informatics, and Mechanics of the University of Warsaw. Janusz Jabłonowski was the contact person for legal issues. Grzegorz Grudziński supervised the project itself.

9.1.2. Communications

Prinz was developed during the Covid-19 pandemic, which influenced the usual development workflow. Except for actions required by law, all meetings were held online.

The team developed the code asynchronously, meeting every week for a sprint retrospective and discussing new issues. They used Slack to bridge a gap arising from the lack of face-to-face consultations during the week. Each Thursday delegated members met with the supervisor to discuss development progress. Team also held Friday meetings with the commissioning party to encourage the bidirectional flow of information.

9.1.3. Timeline

The project started in October 2020 with an estimated deadline of June 2021. The list below presents important milestones.

[1]https://touk.pl/

1. **October 1, 2020**: Start of development.
2. **January 7, 2021**: Presentation of different approaches to serving machine learning solutions at university seminar. Project will later include integrations for these technologies.
3. **January 27, 2021**: MLflow integration merged to `master`. Start of the PMML subproject.
4. **February 4, 2021**: Demo of the MLflow integration to the client.
5. **March 21, 2021**: Start of the H2O subproject.
6. **April 15, 2021**: PMML integration merged to the main branch.

9.2. Development

9.2.1. Tools and technologies

Prinz is a set of extensions for Nussknacker. It adopts similar concepts as the parent project to ease integration and developer onboarding.

Core parts of the solution are Scala 2.12 code built with the `sbt` tool. External libraries often supply only a Java API, which requires the occasional use of Java classes. Example models (along with some adapter code for the serving endpoints) were prepared in Python 3 due to its popularity among the data science community [10].

The team uses `.git` as the project's control version system. Files are hosted remotely on GitHub[2], which also provides tools to manage issues, pull requests, and project boards. While there are no requirements regarding developer tools, all members opted for using IntelliJ IDEA, an IDE often used by Scala engineers. Documentation and specifications use Markdown where possible.

9.2.2. Programming workflow

Development processes in the project rely heavily on the features provided by GitHub.

For each new issue, the engineer creates a development branch. Branch names follow naming convention `username/feature-name`. The developer uses this branch as a development area. Once the feature is ready, they create a pull request to the main project branch.

Pull request is a final stage before merging changes to the code base. Its creation triggers several automated checks (called "GitHub workflows") to enforce code quality and reveal regressions. Code quality evaluation includes simple checks (e.g. for extra white space) and static analysis tools like `scalastyle`. Regression testing comprises unit tests of the project.

Other team members can review the code and request changes. The developer usually pushes new commits with the updates (which in turn triggers workflows again). They mark the conversation as resolved and ask team members for approval. With the approval and all checks passing, the developer checks in their code to the main branch.

The team introduced `dev/feature` branches at a later stage of the project. These are dedicated to complex changes involving multiple contributors. Pushing to `dev/feature` branches follow the same workflow.

[2]https://github.com/

9.2.3. Interesting issues

Protected branches. To prevent accidental changes, `master` branch was marked as protected. This means that contributors can't push to `master` directly. Also, pull requests require approval and passing checks and cannot be merged otherwise. The same rules apply to `dev/feature` branches.

Additional documentation. Project includes additional documentation served as a GitBook[3] web page. Manual pages are written in Markdown and stored along with the code. This approach allows to automate building and deploying HTML pages using existing solutions. Choice of GitBook and its version follows on from the parent project.

[3]https://gitbook.com/

Chapter 10

Responsibilities

There were four students involved in the project. Workload was divided as described:

- Krzysztof Antoniak: setup base repository structure, setup Scalastyle rules, setup sample enricher, add source and sink to sample, setup first FraudDetection model sample, create PMML sample models, setup GitBook, implement `Model` and `ModelInstance` for PMML and H2O,
- Michał Jadwiszczak: design Prinz Model API, setup ngnix, separate structure of `docker-compose.yaml` for every integration, implement `ModelRepository` for PMML and H2O, propose `RepositoryClient` abstraction,
- Jan Kukowski: implement data structures, create MLflow data converter, write unit and integration tests for data structures and conversion, combine Nussknacker types with model signature, implement PMML and H2O signature parsing,
- Maciej Procyk: setup MLflow and Nussknacker to first `dev-environment`, setup files linters, add S3 client, initialize integration tests structure, create configuration structure, introduce HTTP client for MLflow integration, configure automatic publishing of library.

Appendix A

Disc contents

Attached CD contains snapshot of the repository. Its directory structure is presented below with short descriptions of selected nodes.

- `.github` GitHub automation definitions
- `dev-environment` Development environment configuration and specifications
  - `README.md`
  - `clear_environment.sh`
  - `create_environment.sh`
  - `docker-compose-env.yaml` Configuration of Nussknacker environment
  - `docker-compose-h2o.yaml` Configuration of H2O environment
  - `docker-compose-mlflow.yaml` Configuration of MLflow environment
  - `docker-compose-pmml.yaml` Configuration of PMML environment
- `docs` Sources of documentation pages
- `prinz` Prinz API specification
- `prinz_h2o` Implementation of H2O integration
- `prinz_mlflow` Implementation of MLflow integration
- `prinz_pmml` Implementation of PMML integration
- `prinz_proxy` Implementation of models proxying
- `prinz_sample` Example Prinz library usage
- `priz_util` Utility HTTP client
- `project`
  - `build.properties`
  - `plugins.sbt`
  - `scalastyle_config.xml`
  - `scalastyle_test_config.xml`
- `.env` Environment constants defitnition
- `.gitignore`
- `LICENSE`
- `README.md`
- `build.sbt` Scala project definition with dependencies
- `sbtwrapper` sbt standalone compile script

Bibliography

[1] Apache Flink: What is Apache Flink? — Architecture, 2021. `https://flink.apache.org/flink-architecture.html`.

[2] Data Mining Group, 2021. `http://dmg.org/`.

[3] Data Mining Group - PMML Powered, 2021. `http://dmg.org/pmml/products.html`.

[4] How to Use AI and Machine Learning in Fraud Detection, 2021. `https://spd.group/machine-learning/fraud-detection-with-machine-learning/`.

[5] Java PMML API, 2021. `https://github.com/jpmml`.

[6] MLflow – A platform for the machine learning lifecycle, 2021. `https://mlflow.org/`.

[7] MLflow documentation – MLflow 1.17.0 documentation, 2021. `https://mlflow.org/docs/1.17.0/index.html`.

[8] MOJO quickstart: Benefits of MOJOs over POJOs, 2021. `http://docs.h2o.ai/h2o/latest-stable/h2o-docs/mojo-quickstart.html#benefits-of-mojos-over-pojos`.

[9] What is Event Stream Processing? How & When to Use It, 2021. `https://hazelcast.com/glossary/event-stream-processing/`.

[10] Srinath, K. Python – the fastest growing programming language. *International Research Journal of Engineering and Technology 4*, 12 (2017), 354–357.

[11] Zaharia, M. Introducing MLflow: an Open Source Machine Learning Platform, 2021. `https://databricks.com/blog/2018/06/05/introducing-mlflow-an-open-source-machine-learning-platform.html`.