Data Analysis on Machine Learning performance

Objectives

The objective of this project is to define a prototype in C# that is able to analyze in real time the diagnostic of the AIFP system, extending it. The system is able to use different Machine Learning models at run time and, by this way, we are able to analyze the different models and understand what is the best.

We have already defined three different Machine Learning contexts and for each we have one or more Machine Learning models we can analyze.

We are going to use OpenTelemetry and ElasticSearch to provide data analysis.

Methodology and technology being used

We can divide the system into 2 parts: the first is the AIFP system that is the core used to define some ML pipeline and models, and the second is the one related to collecting telemetries about the system.

In order to implement the diagnostics we used OpenTelemetry, that help us to instrument, generate, collect and export telemetry data (metrics, logs, and trances) and ElasticSearch (Elastic, Kibana and APM) to analyze the software's performance and behavior.

In particular:

- *OpenTelemetry SDK* .NET library integrated into the AIFP system;
- *OpenTelemetry Collector* used to collect, process and send telemetries data;
- *Elastic Stack* is composed of *Elastic* (engine - store and analysis), *Kibana* (graphical user interface to visualize and define analysis) and *APM server* (receives data from Elastic APM agents).

Technical implementation

The system architecture is composed of: the system Core, the OpenTelemetry agent integrated into the Core, the OpenTelemetry Collector and ElasticSearch.

The system workflow begins with the data production by the Core during the computational operations as *Fit*, *Predict* and *Evaluate* tasks. These data are collected by the OpenTelemetry Agent which sends it to the OpenTelemetry Collector through OTLP protocol.
The Collector offers a vendor-agnostic implementation on how to receive, process, and export telemetry data. It is made up of the following components:

- **Receivers**: Receivers are used to get data into the collector. You can use it to configure ports and formats the collector can take data in. It could be Push or Pull-Based. You can receive data in multiple formats.
  It has a default *OTLP Format*, but you can also receive data in other popular open-source formats like *Jaeger* or *Prometheus*.

- **Processors**: Processors are used to do any processing required on the collected data like data Massaging, data Manipulation, or any change in the data as it flows through the collector. It can also be used to remove PII data from the collected Telemetry Data, which can be very useful. You can also do things like *batching the data before sending it out*, *retrying in case of exporting fails*, *adding metadata*, *tail-based sampling*, etc.

- **Exporters**: Exporters are used to exporting data to a backend analysis tool. You can send out data in multiple data formats.

With the combination of these Three Components *(Receivers, Processor* and *Exporter)*, OpenTelemetry Collector can be used to build *Data Pipelines*: receiving data in one format, processing it and then sending out the data in another data format.

At the end of the workflow, data is exported in the backend analysis tool that in this case is Elasticsearch where a few charts have been drawn up in order to produce insights.

OpenTelemetry Protocol - OTLP. As already said, the AIFP system and the OpenTelemetry Collector, and the Collector and Elastic APM communicate using the OpenTelemetry Protocol (OTLP) that is a general-purpose telemetry data delivery protocol designed in the scope of the OpenTelemetry project. OTLP is a request/response style protocol: the clients send requests, the server replies with corresponding responses. At writing time the procolo supports gRPC and HTTP 1.1 channels. We use it with gRPC (OTLP/gRPC). The exchanged data is in binary format and it is also going to be implemented in a JSON format.

At the end, what a trace look like in Elastic is:

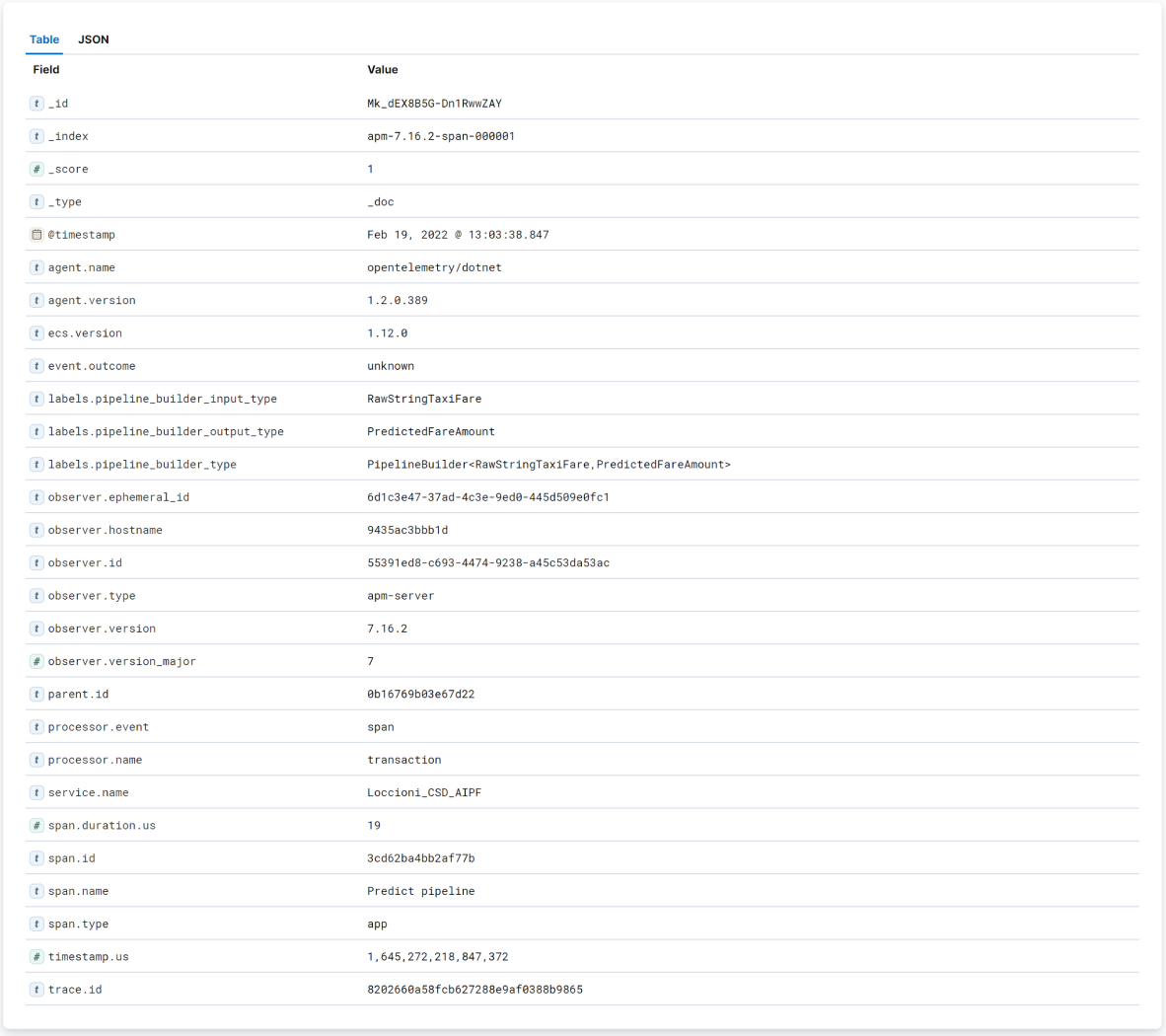


Table JSON

Field	Value
t _id	Mk_dEX8B5G-Dn1RwwZAY
t _index	apm-7.16.2-span-000001
# _score	1
t _type	_doc
@timestamp	Feb 19, 2022 @ 13:03:38.847
t agent.name	opentelemetry/dotnet
t agent.version	1.2.0.389
t ecs.version	1.12.0
t event.outcome	unknown
t labels.pipeline_builder_input_type	RawStringTaxiFare
t labels.pipeline_builder_output_type	PredictedFareAmount
t labels.pipeline_builder_type	PipelineBuilder<RawStringTaxiFare,PredictedFareAmount>
t observer.ephemeral_id	6d1c3e47-37ad-4c3e-9ed0-445d509e0fc1
t observer.hostname	9435ac3bbb1d
t observer.id	55391ed8-c693-4474-9238-a45c53da53ac
t observer.type	apm-server
t observer.version	7.16.2
# observer.version_major	7
t parent.id	0b16769b03e67d22
t processor.event	span
t processor.name	transaction
t service.name	Loccioni_CSD_AIPF
# span.duration.us	19
t span.id	3cd62ba4bb2af77b
t span.name	Predict pipeline
t span.type	app
# timestamp.us	1,645,272,218,847,372
t trace.id	8202660a58fcb627288e9af0388b9865

and a metric look like:

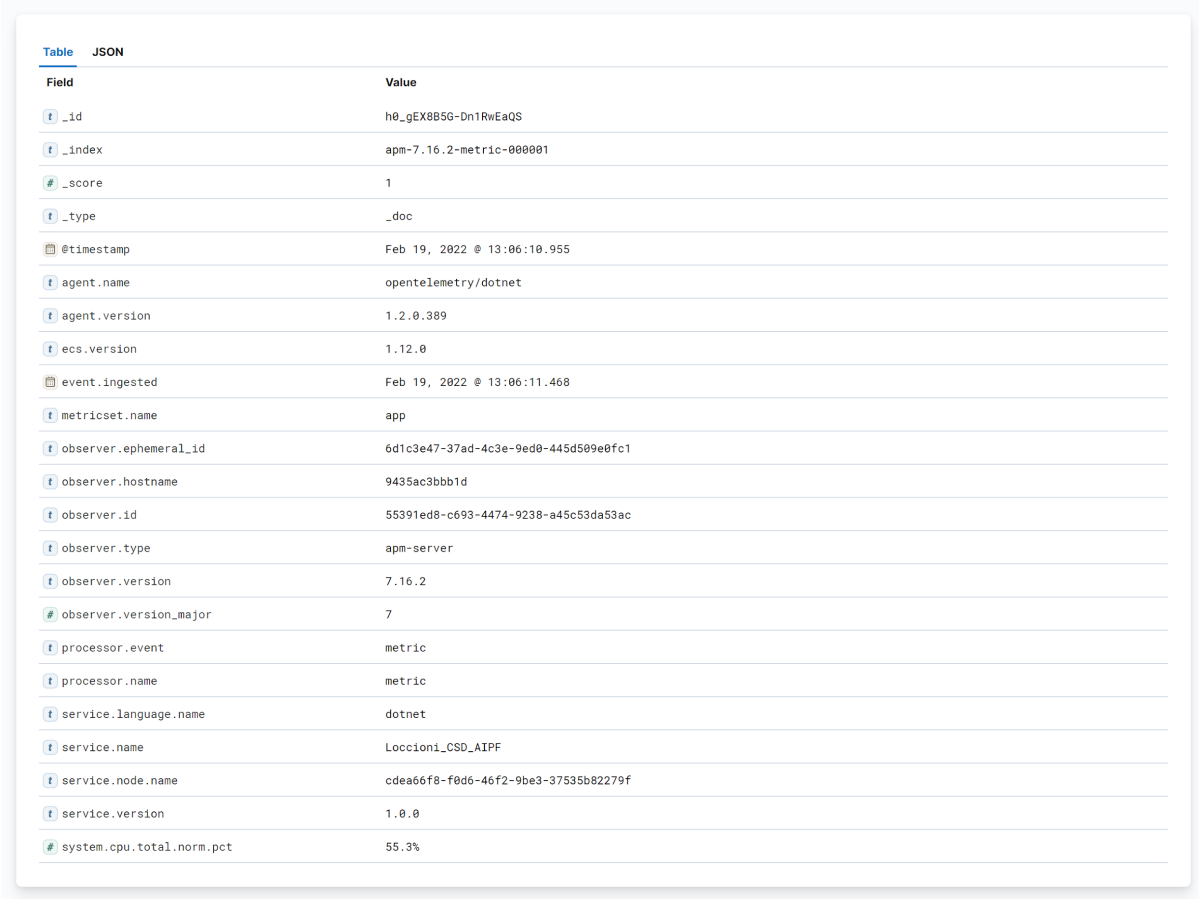


Table JSON

Field	Value
t _id	h0_gEX8B5G-Dn1RwEaQS
t _index	apm-7.16.2-metric-000001
# _score	1
t _type	_doc
@timestamp	Feb 19, 2022 @ 13:06:10.955
t agent.name	opentelemetry/dotnet
t agent.version	1.2.0.389
t ecs.version	1.12.0
event.ingested	Feb 19, 2022 @ 13:06:11.468
t metricset.name	app
t observer.ephemeral_id	6d1c3e47-37ad-4c3e-9ed0-445d509e0fc1
t observer.hostname	9435ac3bbb1d
t observer.id	55391ed8-c693-4474-9238-a45c53da53ac
t observer.type	apm-server
t observer.version	7.16.2
# observer.version_major	7
t processor.event	metric
t processor.name	metric
t service.language.name	dotnet
t service.name	Loccioni_CSD_AIPF
t service.node.name	cdea66f8-f0d6-46f2-9be3-37535b82279f
t service.version	1.0.0
# system.cpu.total.norm.pct	55.3%

Achieved results

Using this system we are able to understand how much time required the system to perform the typical activities of a ML model (Train, Predict and Evaluate the model), the number of request per second and the latency, the number of error that occurs, the usage of CPU and Memory, and other particular analysis.

In general we can say that we are going to analyze the model's performances.

Possible future improvements

The integration of OpenTelemetry with our system is limited because the compatibility of the framework with the Language .NET is partial. Consequently we are not able to use its full potential.

In particular at writing time, the last stable version of OpenTelemetry is v1.1.0 and it does not support logs and metrics but only traces. Moreover the integration of the

OpenTelemetry Protocol is not fully implemented by ElasticSearch, and the main consequence limitations are inability to see host metrics in Elastic Metrics Infrastructure, the logs are not yet supported, inability to view the “Time Spend by Span Type” and inability to see the Stack traces in span/span events.

However we used the beta version of OpenTelemetry v1.2.0 in order to be able to collect some basic metrics (like the time spend by the CPU or the usage of the memory) but they are not well displayed in ElasticSearch; some further improvements are:

- Add logs inside the AIFP system;
- Add metrics about the states of the pipeline (ex: progress indicator);
- Add more specific metrics about CPU, Memory, Network…

Manual

Prerequisite

As already said, the system is developed in the .NET environment and in order to run the application we need the *.NET framework*.

Since we are going to perform analysis on top of this system we are going to use a virtual machine in which we set up OpenTelemetry Collector and the Elastic Stack. The Elastic Stack Docker contains Docker images for all of the products in the Elastic Stack. So we used Docker compose to easily get the default distributions of Elasticsearch, Kibana, and APM Server up and running in Docker.

Configuration

Ubuntu server 20.04.3 LTS
Docker 20.10.10
Docker Compose 1.24.0
Elastic 7.16.2
OpenTelemetry Collector 0.45.0

Minimum RAM for VM: 4GB (Recommended 8GB)

Installation

The first thing to do is to define an ubuntu server (virtual machine), with Docker and Docker-Compose already installed, and transfer the zip "AIPF-Server config.zip" that contains 2 folders:

- *elastic-stack* in which there are:
    - *.env*: a configuration file in which we set the password of elastic and kibana, the version of the stack, the ports of elastic and kibana, the cluster name, the license and finally the memory limiter.
    - *docker-compose.yml*: the docker file of elastic-stack.
- *collector* (GitHub) in which there are:
    - *config.yaml*: the configuration file for the OpenTelemetry Collector that reports the receiver, processor and exporters. In particular, inside the exporter we have to configure the section *headers*, but we will see it later on.
    - *otelcol*: the OpenTelemetry collector executable itself.
    - *LICENSE & README.*

Once you have unzipped "AIPF-Server config.zip", you can modify the *.env* file, in which you can modify the *ELASTIC_PASSWORD* (used to login into ElasticSearch later on) and *KIBANA_PASSWORD* (you have to set a password at least length 6), the *STACK_VERSION* (you have to set an available elastic version, that you can took from here), and other settings. Of course you can leave the default values.

Then you can start the docker-compose using the following command (to issue inside the folder that contains the docker-compose.yml):

```
$ docker-compose up
```

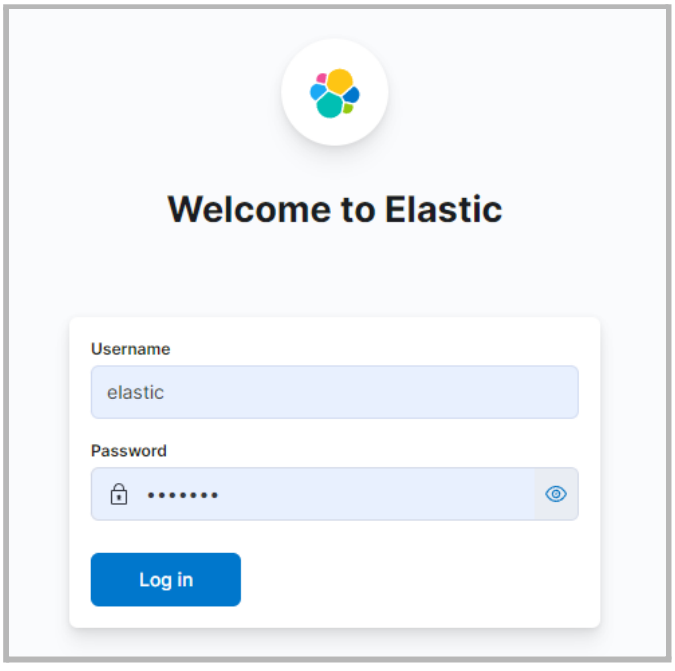
You can use also the command

```
$ docker logs -f
```

to see the logs generated by the docker. At the end you should have something like this:

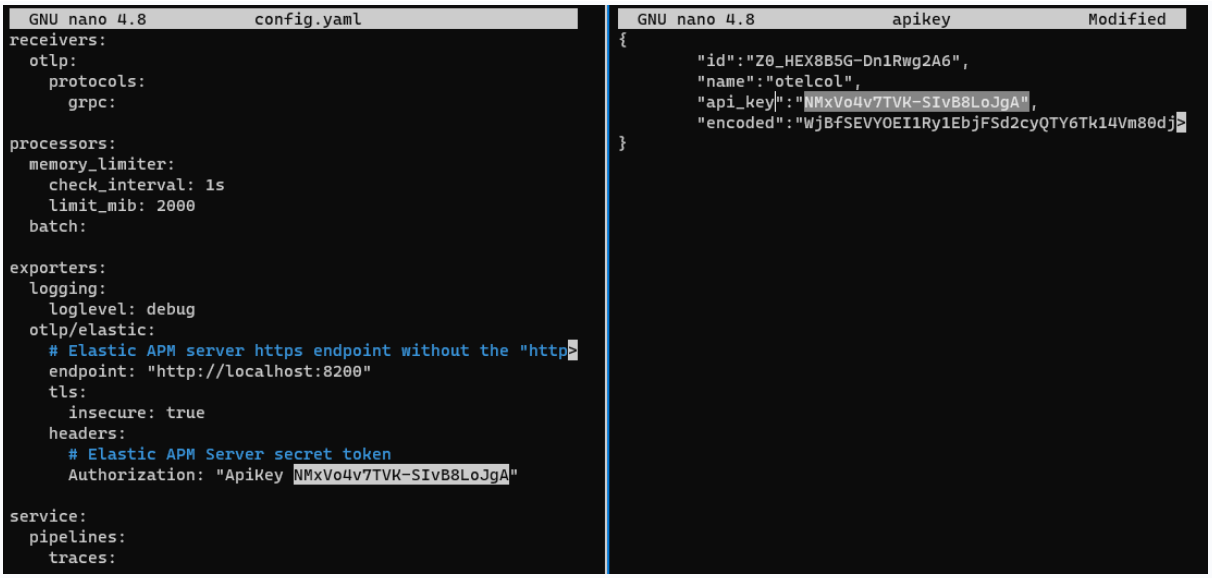
```
data-analytics@data-analytics:~/elasticstack-docker$ docker-compose up
Creating network "elasticstack-docker_elastic" with driver "bridge"
Creating volume "elasticstack-docker_esdata" with local driver
Creating elasticstack-docker_elasticsearch_1 ... done
Creating elasticstack-docker_setup_1         ... done
Creating elasticstack-docker_kibana_1        ... done
Creating elasticstack-docker_apm-server_1    ... done
Attaching to elasticstack-docker_elasticsearch_1, elasticstack-docker_setup_1, elasticstack-docker_kibana_1, elasticstac
k-docker_apm-server_1
elasticsearch_1  | Created elasticsearch keystore in /usr/share/elasticsearch/config/elasticsearch.keystore
elasticsearch_1  | {"type": "server", "timestamp": "2022-02-19T11:32:34,968Z", "level": "INFO", "component": "o.e.n.Node
", "cluster.name": "docker-cluster", "node.name": "85810a90afe9", "message": "version[7.16.2], pid[6], build[default/doc
ker/2b937c44140b6559905130a8650c64dbd0879cfb/2021-12-18T19:42:46.604893745Z], OS[Linux/5.4.0-100-generic/amd64], JVM[Ecl
ipse Adoptium/OpenJDK 64-Bit Server VM/17.0.1/17.0.1+12]" }
```

Now you can connect to elastic using the following url http://server_ip:KIBANA_PORT for example http://192.168.1xx.1xx:5601. You should see the elastic login:



You need to authenticate to the elastic portal using the elastic account (username: elastic, password: ELASTIC_PASSWORD); Once inside, you need to create an Api Key (Stack Management -> Api Key -> Create new ApiKey) that is required in order to send the data from the OpenTelemetry Collector. Save the output as JSON!

So, the next step is to modify the *config.yaml* inside the telemetry folder. In particular you need to override the ApiKey value, under the *headers* section, using the api_key value from the JSON:



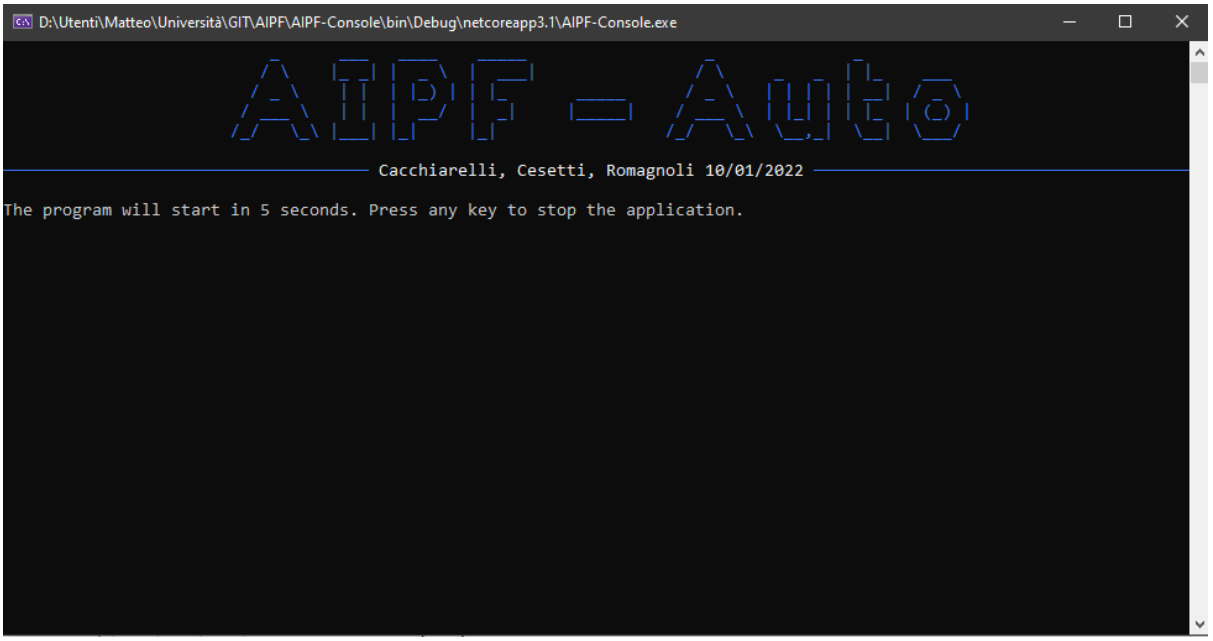
Now you are ready to start the OpenTelemetry Collector using the command:

```
./otelcol --config=config.yaml
```

The configuration of the server is done, you can switch to the client. You have to download the .NET project and open the file config.json located inside AIPF-Console solution with an editor in order to set the correct server IP:

```
{
    "ip": "192.168.178.167",
    "consoleType": "auto",
    "rest": false
}
```

Finally, you can compile and run the AIPF-Console solution:



Performing some operations, like fit, predict or evaluate, the system will report the telemetry data to the collector and the to elastic APM.

In order to see metrics inside elastic APM you have to change the metrics indexes (simply go in apm -> metrics -> settings -> indexes and set to metrics-apm*,apm-*).

Inside the APM the traces/metrics are coming and we can analyze them; we can go to *services* under APM and select the AIPF-service.

Usage

First of all we have to start the docker that contains Elastic Stack (ElasticSearch, Kibana e APM Server) by executing *docker-compose up,* into the elastic-stack folder, and starting the OpenTelemetry Collector executable.

Once it is done, we have to power on AIFP-Console on the guest machine and select one of the following models:

- **MNST**: In this example we use our system to implement a machine learning model that allows us to recognize the numbers from an image. For Training we used the MNIST data set that provides us the images in the form of bites arrays. There are two pipelines for this model:
    - The first pipeline, *MNIST default*, is composed by the following steps: [**i**] we convert the input array of size 32x32, made by 0 and 1, into an Image; [**ii**] we resize the image from 32x32 to 8x8; [**iii**] we convert the image into a flat vector of size 64; [**iv**] we finally apply the ML algorithm (SdcaMaximumEntropy) to the processed data.
    - The second pipeline, *MNIST custom*, is composed by the following steps: [**i**] we resize the input array of size 32x32, made by 0 and 1, to 8x8 by summing 4x4 blocks; [**ii**] we convert this matrix into a flat vector of size 64; [**iii**] we finally apply the ML algorithm (SdcaMaximumEntropy) to the processed data.
- **Taxi-Fare**: In this example we use our system to implement a machine learning model that allows us to estimate the taxi fare. There are four pipelines for this model:
    - *Taxi-Fare Huber*: [**i**] the first step is a filter that remove all useless data; [**ii**] after this, it is applied Euclidean Distance that calculate the distance between two point defined in term of geographic coordinates (Latitude and longitude); [**iii**] on this data is applied a filter that remove all data with a distance that doesn't satisfy it; [**iv**] and finally apply the Huber algorithm in ONNX format to the processed data.
    - *Taxi-Fare Linear*: [**i**], [**ii**], [**iii**] are the same; [**iv**] we apply the Linear algorithm in ONNX format to the processed data.

    - *Taxi-Fare Pca Huber*: [**i**], [**ii**], [**iii**] are the same; [**iv**] we apply the Pca algorithm in ONNX format and then we apply the Huber algorithm in ONNX format to the processed data.
    - *Taxi-Fare Pca Linear*: [**i**], [**ii**], [**iii**] are the same; [**iv**] we apply the Pca algorithm in ONNX format and then we apply the Linear algorithm in ONNX format to the processed data.

- **Robot-Loccioni**: In this example we use our system to implement a machine learning model that allows us to understand if the robot is behaving as we expect or if there are some anomalies. In particular this robot is like an arm with 6 axes for movement and is used to build car headlights. The algorithm, starting from the measured currents, calculates which production cycle has been carried out. If the production cycle does not correspond to that actually executed (so the measured currents are very different from the usual ones) then an alarm is reported. The pipeline that has been implemented is composed by the following steps: [**i**] the first step is a filter that remove all row with missing value [**ii**] after this, it is applied a filter that remove all useless data (for example is removed all event with value 0, that represent an error in value acquisition); [**iii**] and finally apply the ML algorithm in ONNX format to the processed data.

For each model there will be the *opentelemetry* option that will perform the model *Fit* activity, then will execute the *Predict* operation in order to simulate a real system behavior and finally the *Metrics* about model accuracy and other parameters will be shown.

The data produced and sent to ElasticSearch by Opentelemetry are relative to two categories:

- **Metrics**: related to CPU and RAM usage;
- **Trace**: related to the executed activity (*Fit*, *Predict*, *Evaluate*);

In Fit activity we defined the following tags: *"model_name"* define the name of the selected model, "*type*" define the type of the pipeline, "*processed_element*" represent the number of the processed elements and finally "*input_type*" and "*output_type*" define the data input and output type. Also all operation inside the model pipeline are described with similar tags; in particular, we have for the Filters the tags "*filter.type*", "*filter.processed_elements*" and "*filter.input.type*" while for the other Transformers we have: "pipeline_builder.type",

“pipeline_builder.processed_elements”, “pipeline_builder.input.type”, and “pipeline_builder.output.type”.

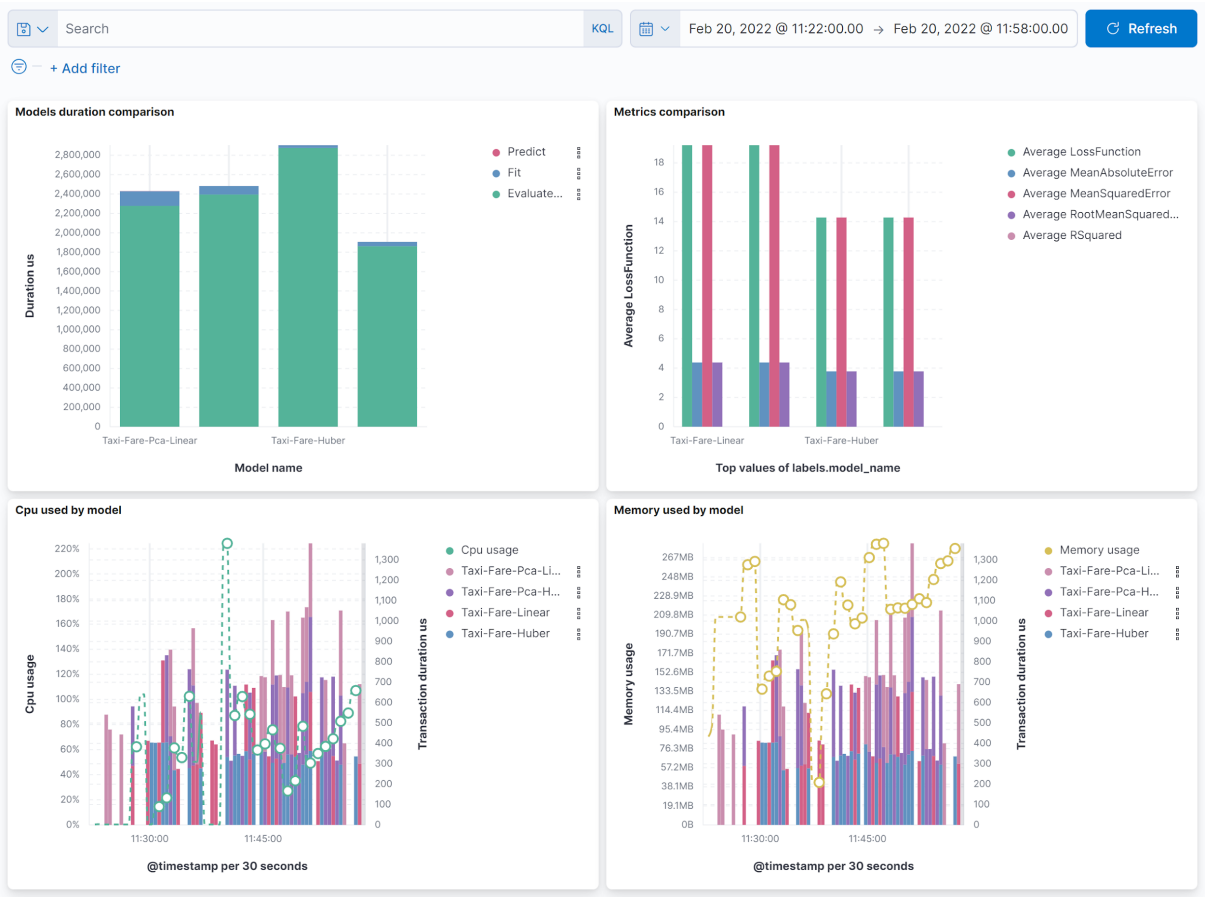
In Predict activity we defined the following tags: *“model_name”* define the name of the selected model, “*type*” define the type of the pipeline, “*processed_element*” represent the number of the processed elements, the “input.[property]” and “output.[property]” that represent the input data and the predicted one and finally “*input_type*” and “*output_type*” define the data input and output type.

In Evaluate activity we defined the following tags: *“model_name”* define the name of the selected model, “*type*” define the type of the pipeline, “*processed_element*” represent the number of the elements used to evaluate the model, “*input_type*” and “*output_type*” define the data input and output type and finally “*metric.[algorithm].[metric_name]*” that represents the metrics of the model and depends on the Machine Learning Algorithm.

Collected data are used to define ElasticSearch's graphs and charts. We defined 3 different dashboards:

1. A general dashboard that contains the CPU and Memory metrics, the model count grouped by name, the models usage during time grouped by name, the average of the activities, and finally charts related to the process and threads.

2. The TaxiFare dashboard in which we compare the 4 models (Linear, Huber, Pca linear, Pca huber); the first graph compare the duration of the main activities (Fit, Predict and Evaluate) for each models, the second one compare the metrics (LogLoss, Mean Absolute Error, Mean Square Error, Root Mean Square, RSquared) of the models, and finally we have the usage of CPU and Memory by each models during time.



3. The MNIST dashboard where the MNIST-Default and MNIST-Custom models are compared; the first graph compare the duration of the main activities (Fit, Predict and Evaluate) for each models, the second one compare the metrics (LogLoss, LogLossReduction, MacroAccuracy, MicroAccuracy, PerClassLogLoss from 0 to 9) of the models, and finally we have the usage of CPU and Memory by each models during time.