University of Camerino

School of Science and Technology

Complex System Design Course

In collaboration with:

LOCCIONI

Artificial Intelligence for FAULT prediction

Students

Damiano Cacchiarelli
damiano.cacchiarelli@studenti.unicam.it

Roberto Cesetti
roberto.cesetti@studenti.unicam.it

Matteo Romagnoli
matteo02.romagnoli@studenti.unicam .it

Project manager

Luca Mazzuferi

Tutor

Matteo Calisti

Supervisors

Prof. Flavio Corradini

Prof. Andrea Polini

AY 2021/2022

Index

Introduction

Application domain

Industrial production and the enormous amount of data produced cannot be managed by hand and to do the data analysis we must use automatic or semi-automatic tools. Analyzing the data coming from the sensors of industrial machines in such a way as to predict possible failures is very important nowadays, because it allows us to improve working conditions, create new business models, increase plant productivity and improve product quality. Many semi-automatic data analysis and management tools can be used, for example the Machine Learning technique used in Big Data Analysis.

In particular, the system in question is based on the integration of Machine Learning algorithms in the data scheme for managing alarms and fault events of the automatic machines of the Loccioni group, a leading company in the sector that designs and creates measurement and control which aim to improve the quality, safety and sustainability of industrial processes and products. The Loccioni group is also known for its willingness to collaborate and to continuously search for human resources, thus maintaining a dynamic and stimulating working environment.

Objective

The team's objective is to model and develop a system, using the ML.NET framework, which allows the integration of Machine Learning algorithms to be applied on data belonging to the Loccioni group.

The presence of a system with these capabilities makes monitoring the machinery present in the company's production chain more efficient, allowing for example to predict any failures or to schedule maintenance activities in the most effective way. So it can be said that the concrete objective of the system is to increase the level of control of the production chain.

Organization of the work

The activities that have been carried out are reported below; in order to improve the organization and distribution of the workload, a GANTT diagram has been defined, which contains the main *Milestones* and related *Deadlines*.

1. **ML.NET Package Analysis**:
    a. Overview *ML.NET*
    b. Analysis of Use Cases
    c. Understanding How You Can Add / Remove an *ML* in *ML.NET*
    d. Analysis of *ONNX* and *TensorFlow* and Integration with *ML.NET Prototype*

2. **Creation (console application) capable of**:
    a. Executing little complex ML algorithms
    b. Extrapolating the data to be analyzed from a local file (JSON, ...)
    c. Importing an *model ONNX*
    d. Define *Unit Case Test* for the implemented functionalities

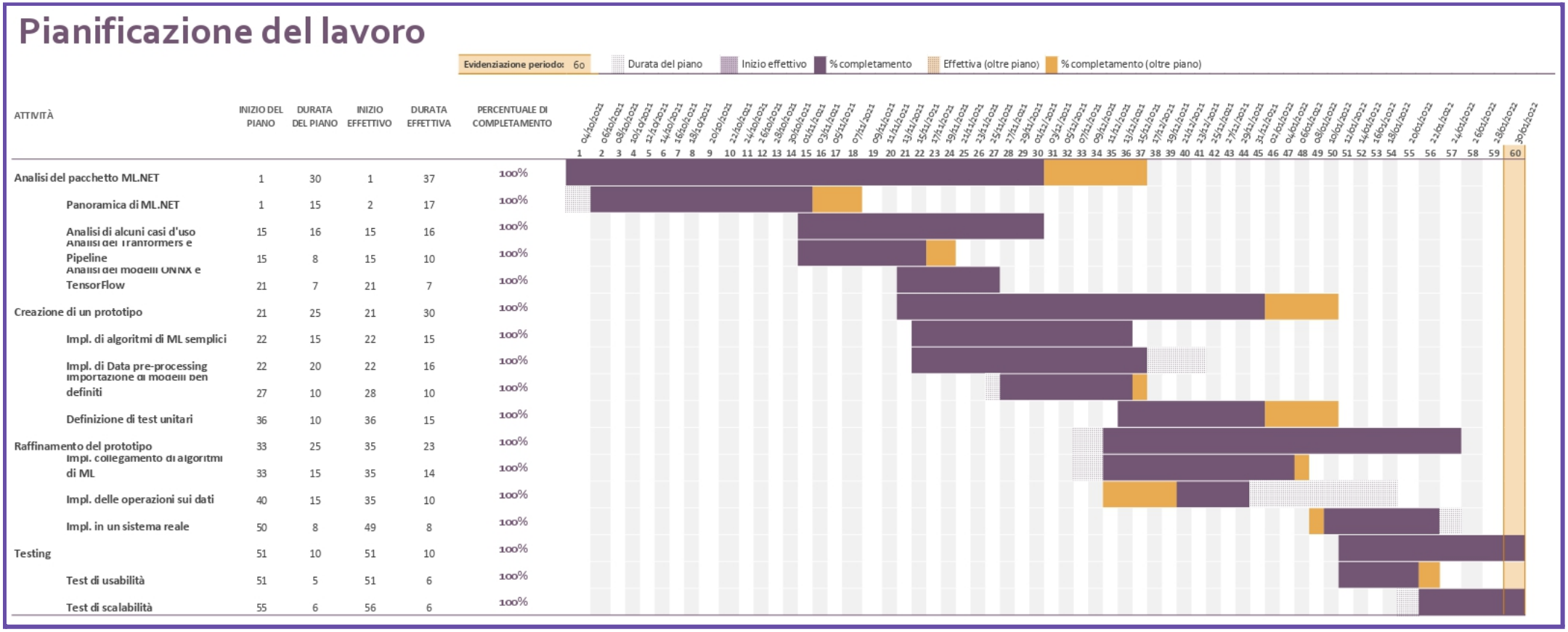
3. **Refine the prototype (REST API application) by adding the functionality**:
    a. Linking of ML algorithms
    b. Operations on data (filter, mapping ...)
    c. Connect the Loccioni database and the existing systems with the developed system
    d. Define *Unit Case Tests* for the implemented functions

4. **Perform system scaling**:
    a. Tests based on the use of large amounts of data
    b. Tests based on the use of increasingly complex ML algorithms

5. **Other possible extensions**:
    a. Add system progress
    b. Improve the velocity and efficiency of the system
    c. Asynchronicity of the data output

The project was developed by adopting an agile approach based on *iterations* weekly in which five activities were carried out: Analysis of the ML.NET package, Modeling, Implementation, Refinement and finally Verification and Testing of the system

Pianificazione del lavoro



ATTIVITÀ	INIZIO DEL PIANO	DURATA DEL PIANO	INIZIO EFFETTIVO	DURATA EFFETTIVA	PERCENTUALE DI COMPLETAMENTO
Analisi del pacchetto ML.NET	1	30	1	37	100%
Panoramica di ML.NET	1	15	2	17	100%
Analisi di alcuni casi d'uso	15	16	15	16	100%
Analisi dei Tranformers e Pipeline	15	8	15	10	100%
Analisi dei modelli ONNX e TensorFlow	21	7	21	7	100%
Creazione di un prototipo	21	25	21	30	100%
Impl. di algoritmi di ML semplici	22	15	22	15	100%
Impl. di Data pre-processing	22	20	22	16	100%
Importazione di modelli ben definiti	27	10	28	10	100%
Definizione di test unitari	36	10	36	15	100%
Raffinamento del prototipo	33	25	35	23	100%
Impl. collegamento di algoritmi di ML	33	15	35	14	100%
Impl. delle operazioni sui dati	40	15	35	10	100%
Impl. in un sistema reale	50	8	49	8	100%
Testing	51	10	51	10	100%
Test di usabilità	51	5	51	6	100%
Test di scalabilità	55	6	56	6	100%

Technologies

The system was developed using the *C#* language program and various frameworks, in particular:

- **ML.NET** Machine Learning library in dotnet environment;
- **Specter Console** support library to enhance the console application;
- **Open Telemetry** system diagnostics library.

Introduction to ML.NET

ML.NET offers the ability to add Machine Learning capabilities to .NET applications, in online or offline scenarios. With this feature, you can make automatic predictions using the data available to your application. Machine learning applications use models in the data to make predictions rather than having to be explicitly programmed.

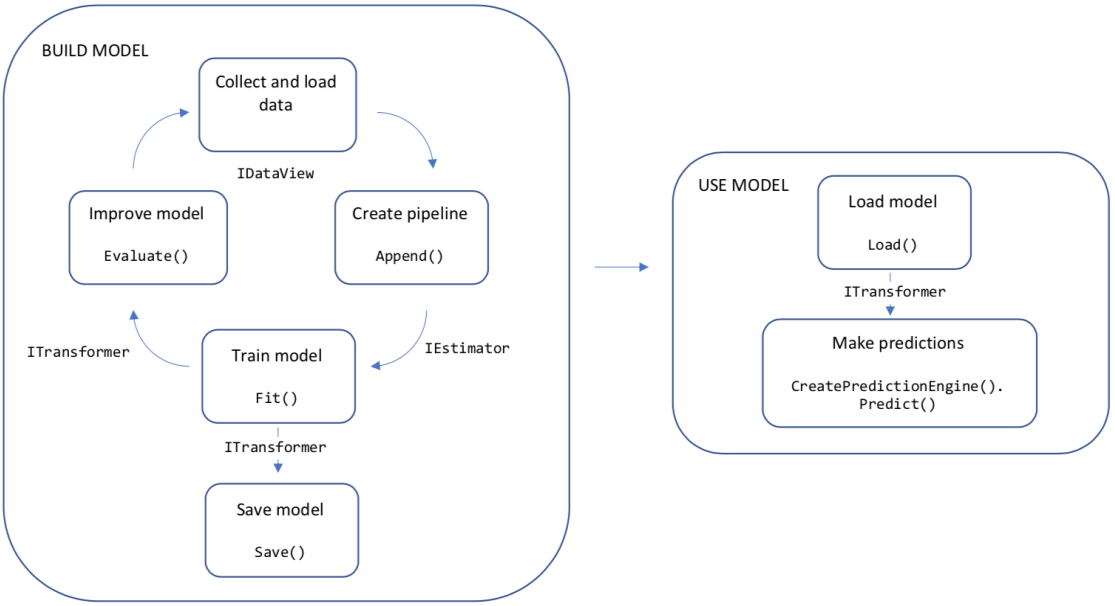
model is fundamental in ML.NET **Machine Learning.**The model specifies the steps required to transform the input data into an estimate. With ML.NET, you can train a custom model by specifying an algorithm or import **TensorFlow** and **ONNX** .
Once you have created a model, you can add it to your application to make predictions.

Machine Learning Model

The following diagram represents the structure of the application code and the iterative process of developing the model:

- Collect and load training data into a IDataView
- Specify a pipeline of operations to extract features and apply a machine learning algorithm
- Train a model by calling **Fit ()** on the pipeline
- Evaluate the model and iterate to improve
- Save the model in binary format, for use in an application
- Load the model into a ITransformer

- Make predictions by calling **CreatePredictionEngine.Predict ()**

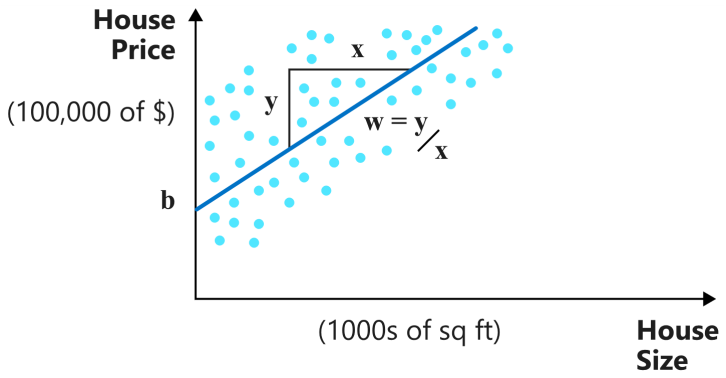


ML model structure

An ML.NET model is an object that contains the transformations to be performed on the input data to obtain the expected output.

Let us consider two models, one simpler and one more complex:

1. The simplest model is two-dimensional linear regression, in which one continuous quantity is proportional to another. The model is simply $Price = b + Size * w$. Parameters *b* and *w* are estimated by fitting a row to a set of pairs (size, price). The data used to find the model parameters is called training **data**. The inputs of a Machine Learning model are characteristics called. In this example *Size* is the only feature. The basic values used to train a Machine Learning model are labels called. In this case the labels are the *Price* in the training set.



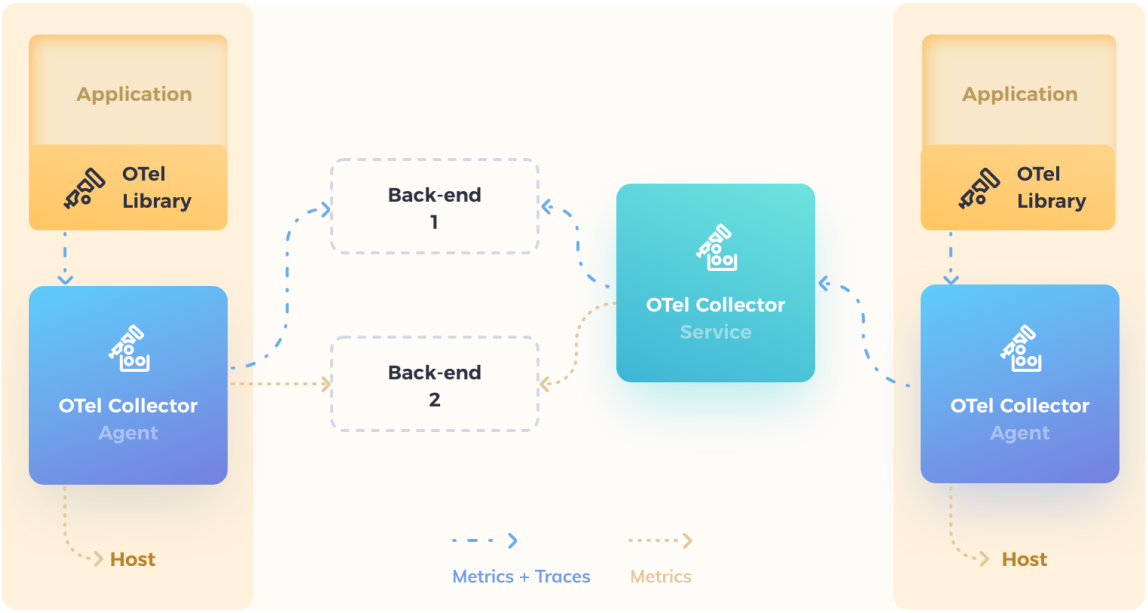
Regression chart example

2. A more complex model classifies financial transactions into categories using the text description of the transaction. Each transaction description is divided into a set of characteristics, by removing redundant words and characters and counting combinations of words and characters. The feature set is used for training a linear model based on the set of categories in the training data. The more similar a new description is to those in the training set, the more likely it is that it will be assigned to the same category.

Both the building pricing model and the text classification model are **linear**. Depending on the nature of the data and the problem to be solved, other models can also be used.

Introduction to OpenTelemetry

OpenTelemetry is a set of API, SDKs, Libraries, and Integrations that is aiming to standardize the Generation, Collection, and Management of Telemetry Data (logs, metrics, and traces).
The data you collect with OpenTelemetry is vendor-agnostic and can be exported in many formats. Telemetry Data has become critical to observe the state of distributed systems.
With microservices and polyglot architectures, there was a need to have a global standard. OpenTelemetry aims to fill that space.



OpenTelemetry architecture

OpenTelemetry Collector

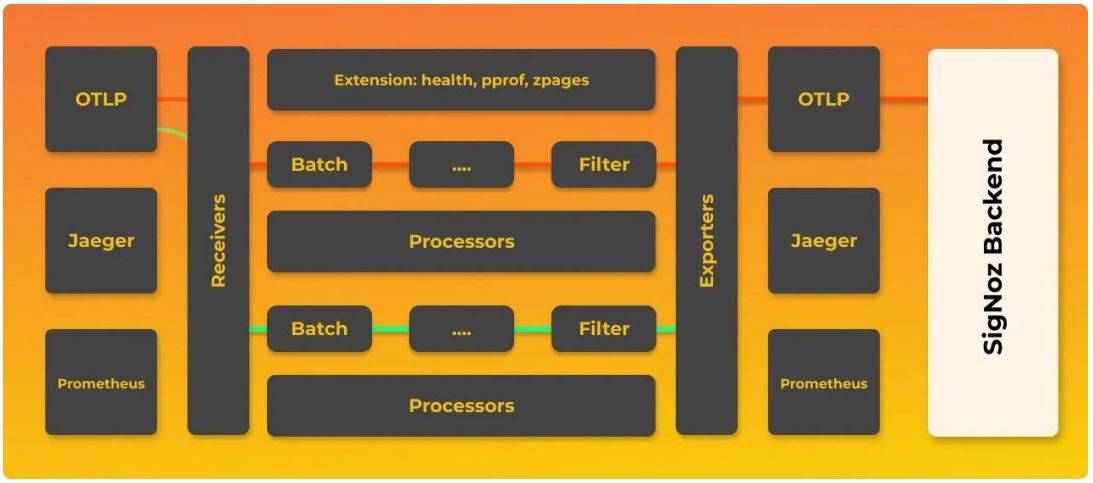
The OpenTelemetry project facilitates the collection of telemetry data via the OpenTelemetry Collector. The OpenTelemetry Collector offers a vendor-agnostic implementation on how to receive, process, and export telemetry data. It removes the need to run, operate, and maintain multiple agents / collectors in order to support open-source observability data formats (e.g. Jaeger, Prometheus, etc.) sending to one or more open-source or commercial back-ends.

The Collector 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 the 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.



OpenTelemetry Collector

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.

Architecture

The final system contains 4 solutions each with specific functionality:

- **AIFP Core** is the kernel of our system and consists of a library that can be added to other existing projects. This solution allows you to integrate Machine Learning functions into other projects.
- **AIFP Console** consists of a console application that uses *AIFP Core* and *AIFP RestController* in order to solve some artificial intelligence problems; the subsystem can be used offline or online (using Rest calls).
- **AIFP RestController** is a service based on Rest calls that uses *AIPF Core* to solve some artificial intelligence problems.
- **AIFP Test** is the solution where there are the tests for *AIPF Core*.

AIFP Core

As already said, this solution represents the kernel of our system and allows us to add Machine Learning functionalities to .NET applications.
To this end, the system has 4 stages of use.

1. **Definition of the pipeline.** In most cases, the available data is not ready for use in training a machine learning model. Raw data must be prepared or *preprocessed* before it can be used to find model parameters by applying one or more *Transformations*, for example:
    a. data may need to be converted from string values to a numeric representation;
    b. or the input data may contain redundant information;
    c. or it may be necessary to reduce or expand the size of the input data;
    d. or the data may require normalization or scaling.
2. **Train the model.** Once the transformation pipeline has been defined, the model needs to be trained using an input dataset. Only after you have done this (which may take several minutes) can you use the template.

3. **Evaluation of the model.** Once the data is prepared, and the model trained, how do you know if future predictions will run correctly? You can evaluate the model against new data defined as test data. Each type of machine learning activity has metrics that are used to evaluate the accuracy and precision of the model against the test dataset.
4. **Predict the data.** Finally, it is possible to make predictions based on the defined model.

The system can be divided into different sections, each with some peculiarities:

- Manager
- Actions
- Metrics
- Telemetry

Manager

The interface for creating a context to perform Machine Learning activities is the *MLManager*; when you want to use the library you need to create an *MLManager <I, O>* specifying the type of input *I* and output *O.*
This class provides several methods that allow you to create a pipeline of transformations and then define a model (in this phase only the creation of the objects is performed; no execution takes place yet so the pipeline is not executed, but only instantiated),

```
public MLBuilder<I, O> CreatePipeline ()
```

train the model (method `Fit ()` uses the input training data to estimate the model parameters; this process is known as model training),

```
public async Task<IDataView> Fit (IEnumerable <I> rawData)
public async Task<IDataView> Fit (IDataView rawData)
```

evaluate the model

```
public async Task <List<MetricContainer>> EvaluateAll (IEnumerable <I>
data)
```

and make predictions using the model (the model transforms the input data in estimates)

```
public async Task<O> Predict (I toPredict)
```

All operations can occur asynchronously.

It also has the property *MLLoader <I>* which allows you to upload files in csv format and transform them into an *IEnumerable <I>* (a list of type *I*) or into *the IDataView*. An important property of the *IDataView*'s object is that they are evaluated deferred: data is loaded and processed only during training, model evaluation, and data prediction.

Actions

Actions are all operations that can be performed in a pipeline.
In particular we have divided the actions into:

- **Filter** operations (*IFilter*): the filter allows you to delete an object (or more than one) based on the predicate; in other words, if the filter is not satisfied the object is deleted.
- **Transformation** (*IModifier*): a modifier represents a transformation of the input data into an output data. Therefore, it is possible to execute a particular algorithm that allows the conversion of data in input of type *I* into data in output of type *O.*
  Data transformations are used to:
  - Prepare data for model training;
  - Apply an imported model in *TensorFlow* or *ONNX format;*
  - Post-process the data after passing through a model.

Each transformation in the pipeline has an input schema (names, types, and sizes of the data that the transformation expects as input) and an output schema (names, types, and sizes of the data that are produced after the transformation).
If the output schema of one pipeline transformation does not match the input schema of the next transformation, an exception is thrown.

The representation of the transformation chain is associated with the *Pipeline*. In order to concatenate the actions without following a particular pattern (example: `Filter => Transformation => Filter => Transformation => Transformation =>...)` classes have been defined as MLBuilder and *PipelineBuilder* which add a higher level of abstraction.

MLBuilder. When the `public MLBuilder<I, O> CreatePipeline ()` method is invoked, the system generates a first *MLBuilber*. This class is a container (wrapper class) for a single action which can be a filter or a set of transformations (*PipelineBuilder*).
This class has the following two methods for concatenating filters and transformation sets:

```
public Pipeline<R, O> AddTransformer<R>(IModifier<I, R> modifier)
public MLBuilder<I, O> AddFilter (IFilterAction<I> filter)
```

In this way you can write for example the following pipeline

```
mlBuilder.AddFilter (filter1)
   .AddTransformer (modifier1)
   .AddFilter (filter2)
   .AddTransformer (modifier2)
  ...
```

PipelineBuilder. A pipeline builder represents a set of transformations. In particular, it is sometimes necessary to apply several transformations one after the other; therefore, this class allows you to concatenate multiple transformations taking into account the types of

input and output (the output of one transformation must correspond to the input of the next transformation). Example:

```
pipelineBuilder.Append<I, O1>(modifier1)
   .Append<O1, O2>(modifier2)
   .Append<O2, O3>(modifier3)
  ...
  .Build ()
```

Once the set of transformations has been defined, to build the pipeline, the `Build ()`.

Metrics

Another component of the system concerns the evaluation of a model. There are two main classes: **MetricContainer** and **MetricOptions**. These classes allow you to define the list of specific metrics for the type of Machine Learning algorithm.
The metrics are available through the method present in MLManager

```
public async Task <List<MetricContainer>> EvaluateAll (IEnumerable <I>
data)
```

According to the machine learning algorithm used, different metrics are available (the most important):

- Accuracy or Precision, F1 score for binary classification algorithms;
- Micro / Macro Accuracy, Log loss for multiclass classification algorithms;
- R2, Absolute loss, RMS-loss, Mean square error for regression algorithms.

In order to view the metrics of a pipeline that uses an ONNX pre-trained model, the following attributes have been defined:

- **EvaluateColumn** decorator for a property; allows to associate the name of the input column with the property.
- **EvaluateAlgorithm** decorator for a class; allows to specify the type of algorithm used and the name of the property in input.

For example, if we refer to the ONNX model used for TaxiFare, in order to evaluate the metrics of the resulting model, it is necessary to define this class:

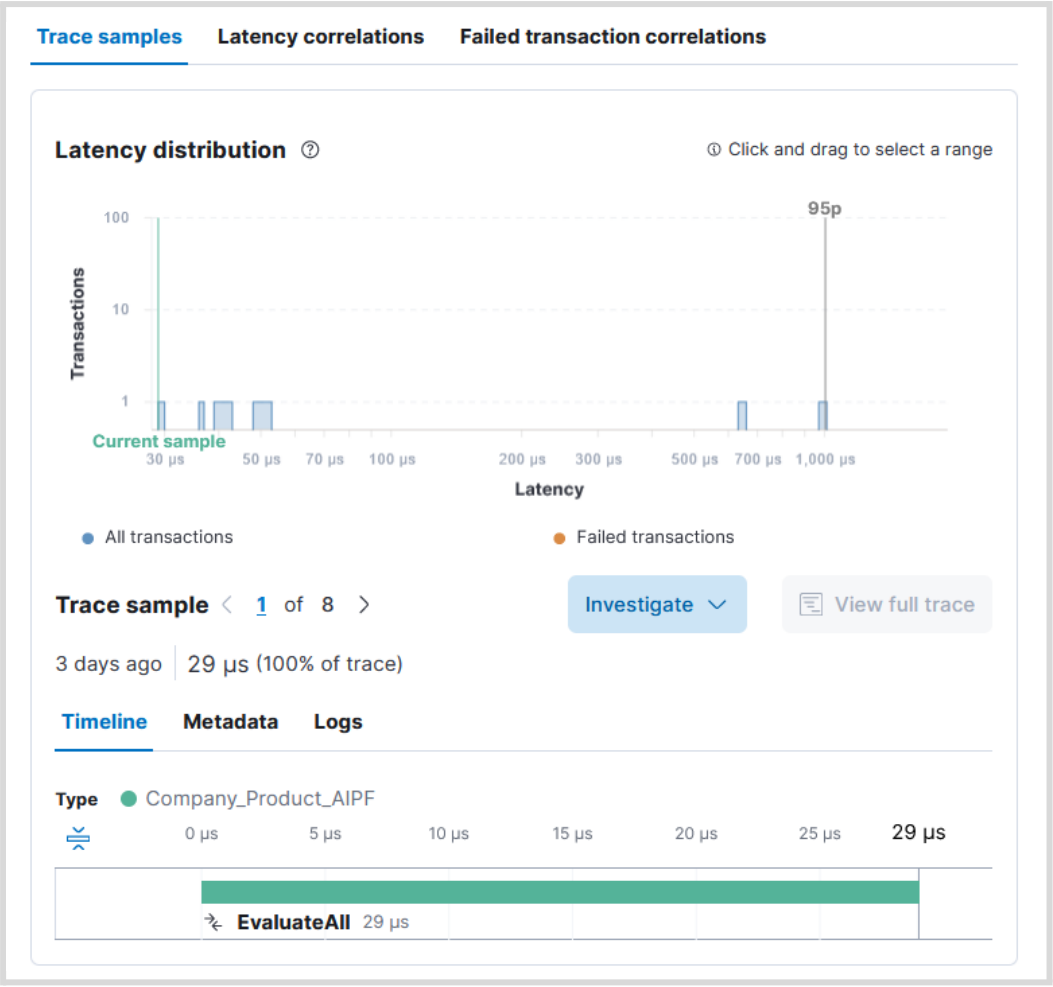
```
    [EvaluateAlgorithm (EvaluateAlgorithmType.REGRESSION,
"labelColumnName", "FareAmount")]
    public class RegressionEvaluate
    {
        [EvaluateColumn ("scoreColumnName")]
        public float PredictedFareAmount { get; set; }
    }
```

Telemetry

The last (additional) component of the system concerns diagnostics: monitoring the system in the form of traces, metrics and logs. For this purpose the **TelemetryTracer** class has been created with the task of creating the *Opentelemetry* component responsible for monitoring activities.

The metrics are sent to the OpenTelemetry Collector and viewed via ElasticSearch. In this way it is possible to analyze the program performance, understand if one model is better than another, highlight some phenomenon, anomaly, etc.

For example, we can see that by running the `EvaluateAll ()`. with an incremental number of samples, the response latency increases (as expected):



Example Elasticsearch

Functionality

Three use cases of the system have been implemented: MNIST, Taxi-Fare and Robot-Loccioni. In each use case, algorithms have been implemented to be applied on the data provided by the tutor of the Loccioni group Matteo Calisti.

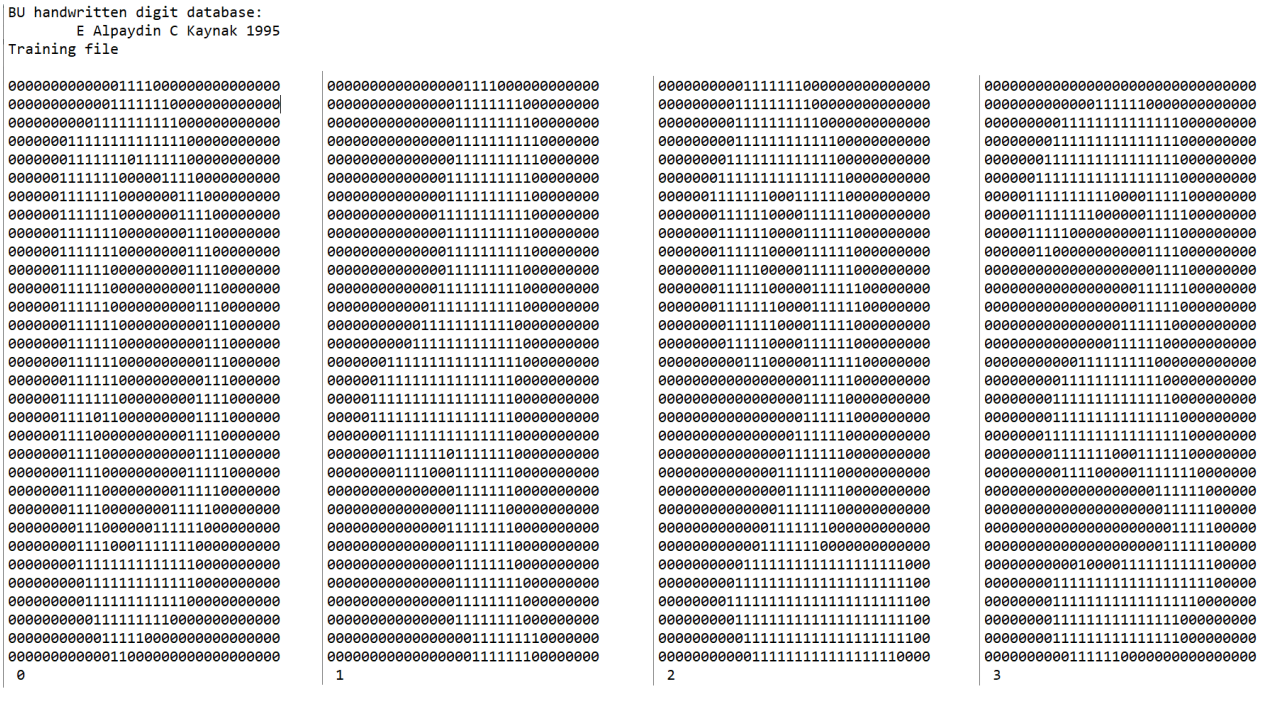
In the first example we were advised to take the data present in the MNIST Database and process them in such a way that they can be used to train the model.

The second use case, on the other hand, was implemented by obtaining data from *kaggle.com*, a site that offers various Machine Learning problems.

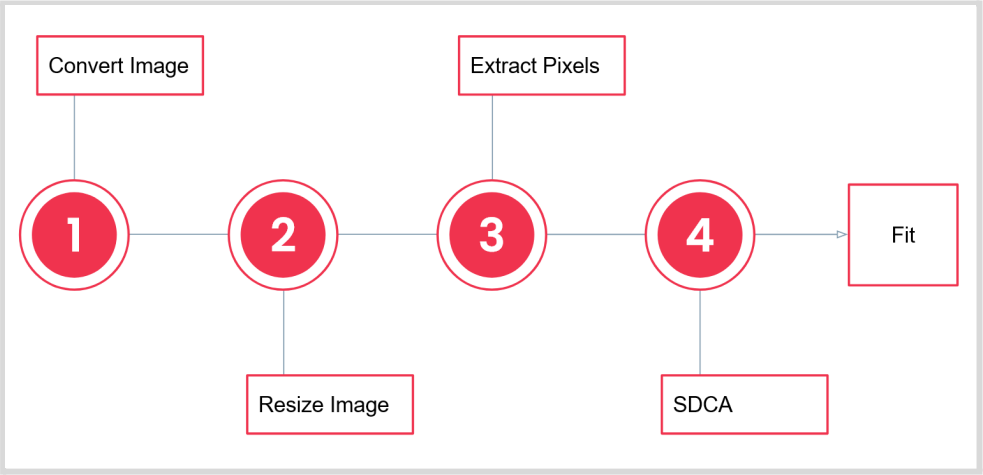
Finally, in the implementation of the last use case, the tutor provided us with data and information about the Loccioni robots present in the production chain. Based on these data, the system is able to understand if the robot is in production, is performing maintenance or if it has any faults.

MNIST

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. The training file is similar to this:



Pipeline



Pipeline MINST

The pipeline that has been implemented is composed by the following steps:

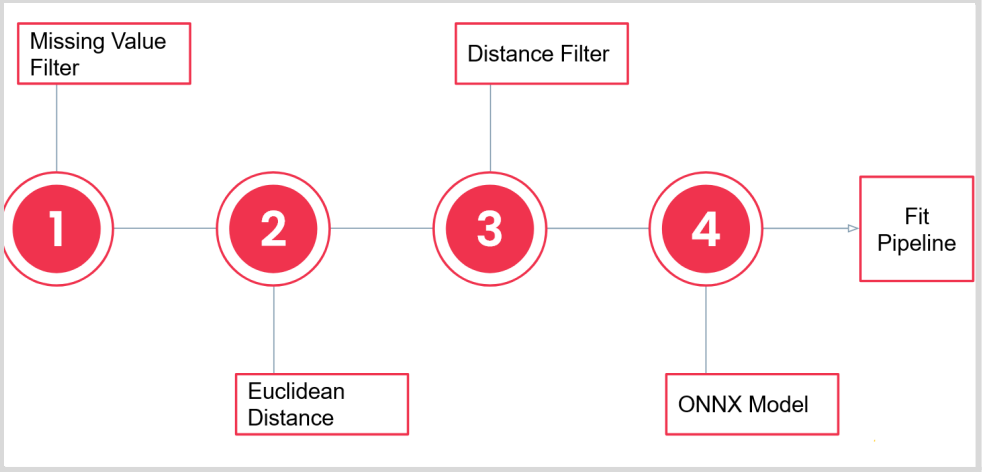
1. we convert the input array of size 32x32, made by 0 and 1, into an Image;
2. we resize the image from 32x32 to 8x8;
3. we convert the image into a flat vector of size 64;
4. We finally apply the ML algorithm (*SdcaMaximumEntropy*) to the processed data.

```
mlManager.CreatePipeline ()
                    .AddTransformer (new
ProgressIndicator<VectorRawImage> ("MNISTProcess # 1"))
                    .Append (new VectorImageResizer())
                    .Append (new SdcaMaximumEntropy(500))
                    .Build ();
```

Taxi-Fare

In this example we use our system to implement a machine learning model that allows us to estimate the taxi fare.

Pipeline



Pipeline Taxi-Fare

The pipeline that has been implemented is composed by the following steps:

1. the first step is a filter that remove all useless data;
2. after this, it is applied Euclidean Distance that calculate the distance between two point defined in term of geographic coordinates (Latitude and longitude);
3. on this data is applied a filter that remove all data with a distance that doesn't satisfy it;
4. and finally apply the ML algorithm in ONNX format to the processed data.

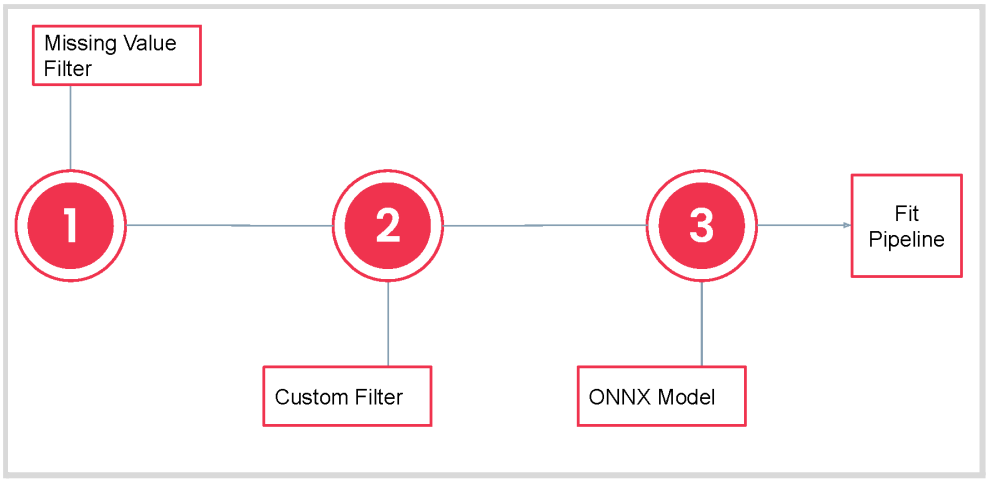
```
mlManager.CreatePipeline ()
                    .AddFilter (new MissingPropertyFilter
<RawStringTaxiFare> ())
                    .AddFilter (i => i.PassengersCount> = 1 &&
i.PassengersCount <= 10)
```

```
                    .AddTransformer (new ProgressIndicator
<RawStringTaxiFare"))
                    .Append (new GenericDateParser <RawStringTaxiFare,
float, MinutesTaxiFare> ("yyyy-MM-dd HH: mm: ss UTC", IDateParser
<float> .ToMinute))
                    .Append (new EuclideanDistance <MinutesTaxiFare)
                    .Build ()
                    .AddFilter (i => i.Distance> 0 && i.Distance <=
0.5)
                    .AddTransformer (new ConcatenateColumn
<ProcessedTaxiFare> ("input", nameof(ProcessedTaxiFare.Date),
nameof(ProcessedTaxiFare.Distance),
nameof(ProcessedTaxiFare.PassengersCount)))
                    .Append (new ApplyOnnxModel <ProcessedTaxiFare,
object> ("OnnxDir / skl_pca.onnx"))
                    .Append (new DeleteColumn <object> ("input"))
                    .Append (new RenameColumn <object> ("variable",
"input"))
                    .Append (new DeleteColumn <object> ("variable"))
                    .Append (new ApplyEvaluableOnnxModel <object,
PredictedFareAmount, RegressionEvaluate> (
                        "OnnxDir / skl_pca_linReg.onnx",
                        (i, o) =>
                        {
                            o.PredictedFareAmount = i.FareAmount [0];
                        }))
                    .Build ();
```

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.

Pipeline



Pipeline Robot

The pipeline that has been implemented is composed by the following steps:

1. the first step is a filter that remove all row with missing value
2. 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);
3. and finally apply the ML algorithm in ONNX format to the processed data.

```
mlManager.CreatePipeline ()
                    .AddFilter (new MissingPropertyFilter<RobotData>
())
                    .AddFilter (i => i.EventType! = 0)
                    .AddTransformer (new ProgressIndicator<RobotData>
("RobotLoccioniProcess # 1"))
                    .Append (new ConcatenateColumn<RobotData>
("float_input", propertiesName))
                    .Append (new ApplyEvaluableOnnxModel<RobotData,
OutputMeasure, MulticlassEvaluate> ("OnnxDir /
current_model_robot.onnx",
                        (i, o) =>
                        {
                            o.PredictedEventType = i.EventType [0];
                            o.ProbabilityEventType = i.Get
                        } )Probability ()
                    . Build ();
```

Conclusion and future developments

To conclude we students are satisfied with the experience lived with the Loccioni company and in particular with the people we interacted with who are Luca Mazzuferi, very welcoming during the visit to the company, and Matteo Calisti who gave us constantly helped during the development of the project.
As far as the system is concerned, we can say that it meets all the requirements that were defined during the meetings with the development team, however it could be improved by adding additional features such as:

- Automate the insertion and removal of algorithms within the system.
- Provide a more generic implementation for the input algorithms so that you don't have to create classes for every single use case.
- Perform asynchronous predictions by showing a result at a time.

These possible extensions will serve to make the system more efficient by optimizing the control activity that is carried out on the production machinery.

Bibliography and Sitography

[1] ML.NET - Documentation
https://docs.microsoft.com/en-us/dotnet/machine-learning/

[2] .NET - Documentation
https://docs.microsoft.com/en- us / dotnet / machine-learning /

[3] Dataset MNIST - yann.lecun
http://yann.lecun.com/exdb/mnist/

[4] Taxi-Fare - kaggle https://www.kaggle.com/c /
new-york-city-taxi-fare-prediction

[5] OpenTelemetry - Documentation
https://opentelemetry.io/docs/

[6] OpenTelemetry Collector - Documentation
https://signoz.io/blog/opentelemetry-collector-complete -guide /

[7] Specter Console
https://spectreconsole.net/