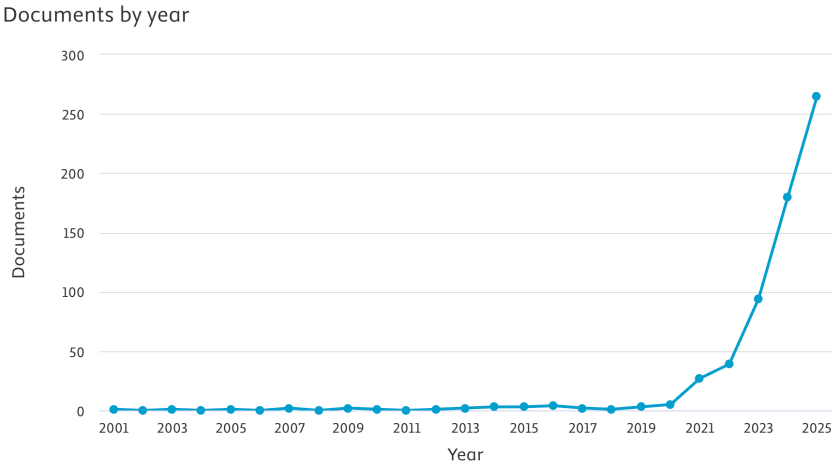
Forgetting Right: Towards Reproducible Benchmarks in Machine Unlearning with ERASURE

Andrea D'Angelo

Università degli Studi dell'Aquila / Italy

Machine Unlearning

Machine Unlearning (MU) is a rapidly growing field, with several publications and methods being published every month.



- A quick scopus search shows that the topic is quickly on the rise.

Rationale (1)

- As often the case for new topics, benchmarking standardization is still lacking.
- We noticed that many works build code only relevant to their specific use case.
- This results in **duplicated efforts** and **low reproduciblity**.

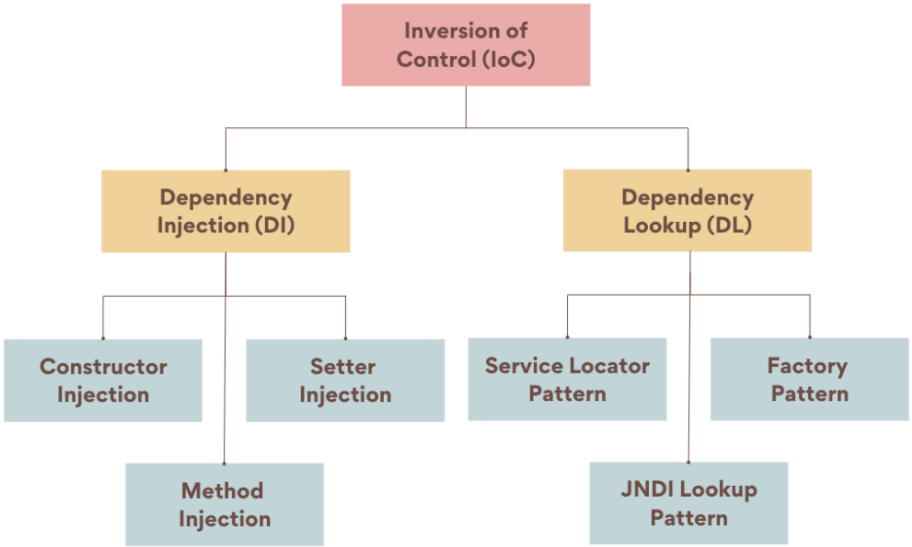
Rationale (2)

- The MU community needs a standardized tool for **reproducible** and **extensible** testing.

- This was our rationale for building **ERASURE**: a modular, extensible framework for Machine Unlearning.

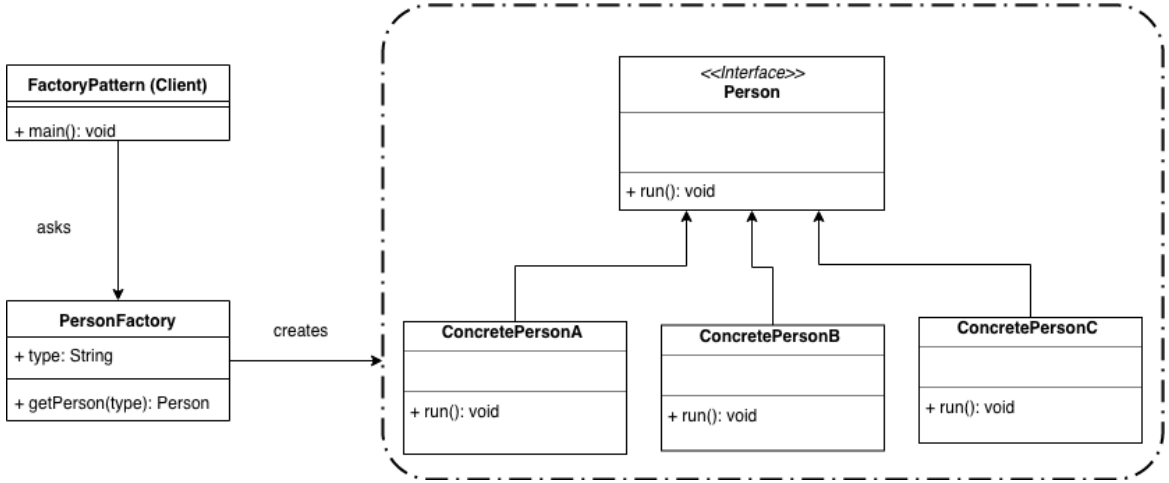
ERASURE is a modular, extensible Machine Unlearning Framework.

Design Patterns (1)



- **Inversion of Control (IoC):** shifts control of object creation and execution flow from client code to an external framework. IoC reduces tight coupling by delegating dependency management to an external provider, often through *dependency injection*. ERASURE leverages IoC for dynamic dependency management.

Design Patterns (2)

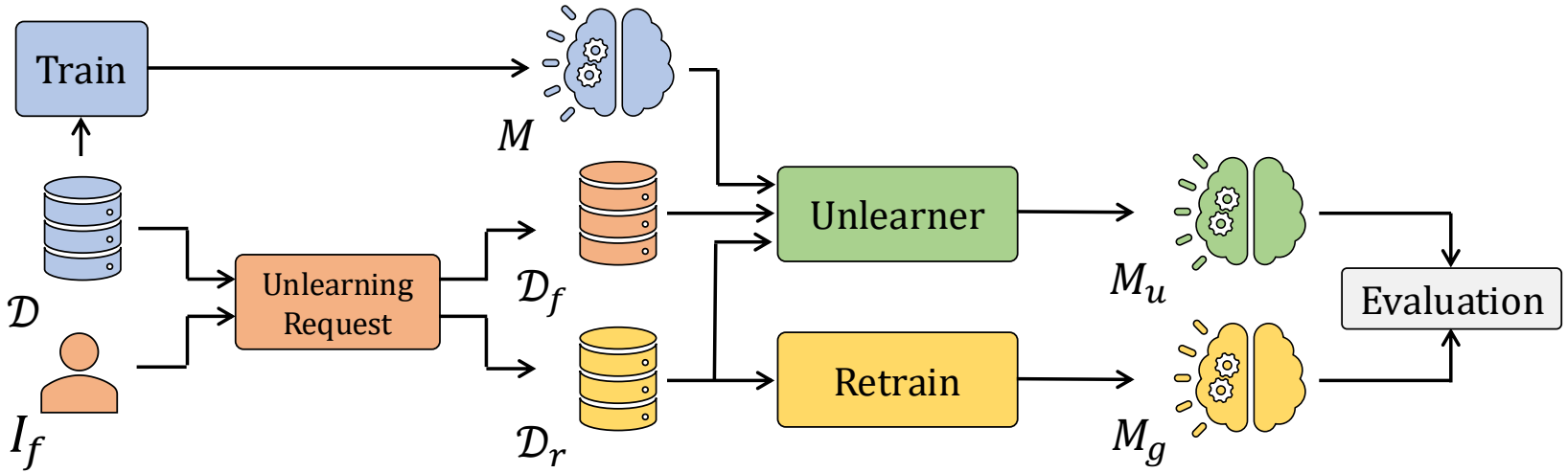


- **Factory Pattern**: abstracts object instantiation. It encapsulates object creation within a dedicated factory class, enabling flexible instantiation. ERASURE uses the Factory Pattern to instantiate Datasets, Models, Unlearners, and everything that is needed for a complete Machine Unlearning workflow.

Machine Unlearning Workflow

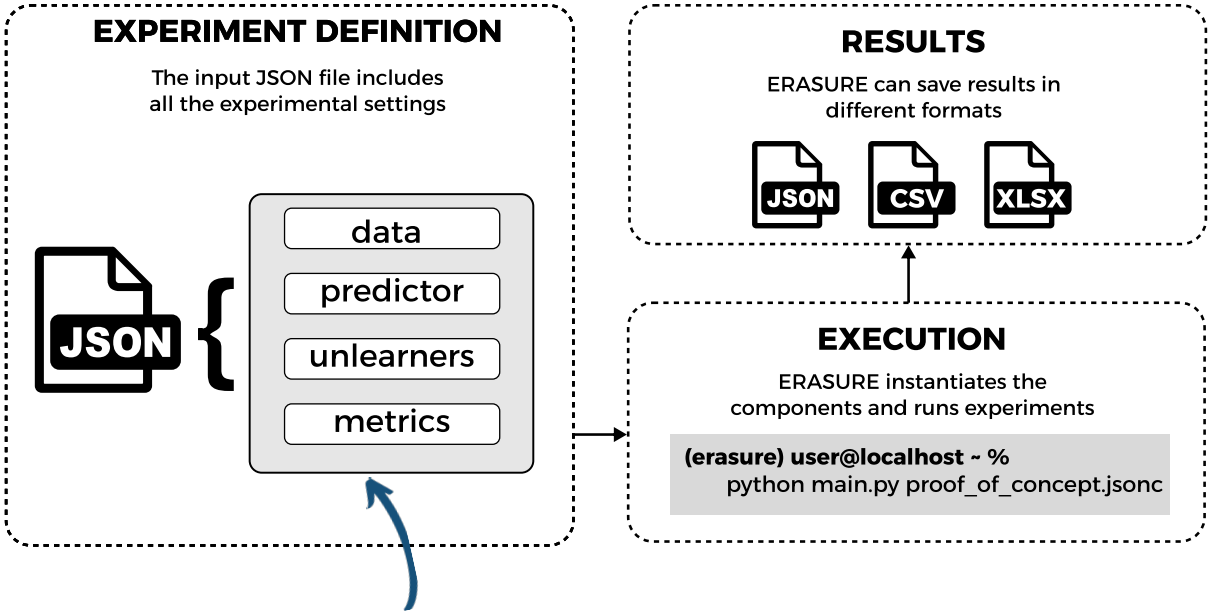
ERASURE was built with the MU workflow at its core.

ERASURE handles all the steps of a typical Machine Unlearning workflow:



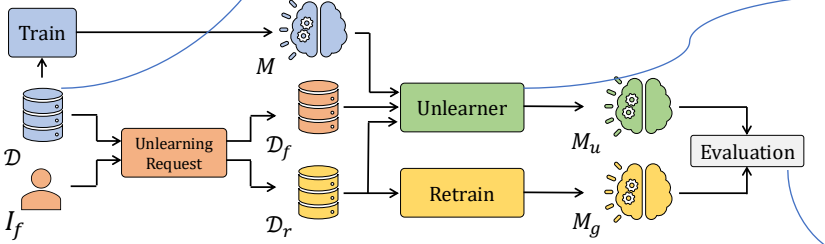
ERASURE's Configuration

ALL the Experimental settings in ERASURE are defined through a single JSON file, called **Main Configuration**.



The Main Configuration file needs to specify all these components.

Main Configuration

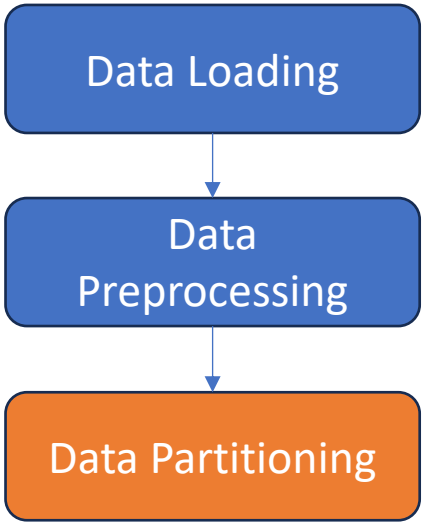



```
"data": {"class":"erasure.data.<NS>.DatasetManager",
      "parameters": {
         "DataSource": { ... },
         "partitions": [ "p_1", "p_2", ... , "p_n" ]
       }},
"predictor": {"class": "erasure.model.<NS>.TorchModel",
 "parameters": {
   "optimizer": {"class": "torch.optim.Adam"},
   "loss_fn": {"class": "torch.nn.CrossEntropyLoss"},
   "model": {"class": "erasure.<NS>.BERTClassifier"}
 }},
"unlearners":[
   {"compose_gold" : "configs/snippets/u_gold.json"},
   .
   .
   ,
   {"class":"erasure.<NS>.AdvancedNegGrad",
      "parameters": {
           "epochs": 1,
           "ref_data_retain": "retain",
           "ref_data_forget": "forget",
           "optimizer": {"class": "torch.optim.Adam",
                "parameters": {"lr": 0.0001}}}
   }
],
"evaluator":{
 "class": "erasure.evaluations.<NS>.Evaluator",
 "parameters": { "measures":[ ... ] } }
```

These are the same modules of the Machine Unlearning workflow.

Data Partitioning

Machine Unlearning requires more demanding Data handling with respect to usual downstream tasks.



ERASURE introduces a flexible DataSplitter strategy, enabling datasets to be partitioned by configuration in virtually unlimited ways.

Each DataSplitter defines a specific partitioning logic, such as sampling a fixed percentage of data or selecting samples from a given class.

Since they are executed sequentially, the DataSplitters can reference previously created partitions.

Data Partitioning (2)

ERASURE also supports a parameter Z.

Imagine you want to remove all the images of George Clooney from CelebA.

There are the labels (Y)

There are the samples (X)

Data Partitioning (3)

There is no information on what celebrity is pictured in each photo! If we only pass the plain dataset (X and Y), we will not be able to filter out pictures, which is critical in Machine Unlearning.

Data Partitioning (4)


```
13    "partitions": [
14       {"class":"erasure.data.<NS>.DataSplitterByZ",
               "parameters":{"parts_names":["forget", "other"],
15              "z_labels":[1]}},
16       {"class":"erasure.data.<NS>.DataSplitterPercentage",
               "parameters":{"parts_names":["retain", "test"],
17              "percentage":0.8, "ref_data":"other"}},
18       {"class":"erasure.data.<NS>.DataSplitterConcat",
               "parameters":{"parts_names":["train","-"],
19              "concat_splits":["retain","forget"]}}
20       ],
```

For instance, the first DataSplitter, called DataSplitterByZ, splits the data in »forget» and «other» based on the passed z_labels, in this case [1].

Intuitively, if the sample has z=1, the sample is put in the forget set.

In green, the cascading feature in effect: the second DataSplitter will split only the data from «other».

Model Training


```
"predictor": {"class": "erasure.model.<NS>.TorchModel",
 "parameters": {
   "optimizer": {"class": "torch.optim.Adam"},
   "loss_fn": {"class": "torch.nn.CrossEntropyLoss"},
   "model": {"class": "erasure.<NS>.BERTClassifier"}
 }},
```

ERASURE handles batched training automatically.

All the training parameters are defined in the configuration file, along with the seed for reproducibility.

ERASURE easily handles classes and parameters from external libraries, like Pytorch or sci-kit learn.

Unlearners

The third section of a JSON Main Configuration file is the “unlearners” JSON Array.

Consider, for instance, the Unlearner called AdvancedNegGrad.

```
"unlearners":[
    {"compose_gold" : "configs/snippets/u_gold.json"},
    .
    .
    ,
    {"class":"erasure.<NS>.AdvancedNegGrad",
       "parameters": {
            "epochs": 1,
            "ref_data_retain": "retain",
            "ref_data_forget": "forget",
            "optimizer": {"class": "torch.optim.Adam",
                 "parameters": {"lr": 0.0001}}}
    }
],
```

It requires a series of parameters, like epochs or optimizer, that can be passed directly from the JSON Configuration file.

Implementing new unlearners

To implement a **Custom Unlearner**, one can extend the **TorchUnlearner** class and implement its __unlearn__(self) method.

It's that simple.

- __unlearn__(self) takes nothing as input but has **access to all the parameters** that were passed through the configuration file, all the dataset and datasets partition, and the original model.
- **The original model is given in isolation**, meaning that all Unlearners has access to their copy. So, the modifications made by one Unlearner will not propagate to other Unlearners.
- The __unlearn__(self) method must **return a modified version of the original model**. This returned model will be evaluated lately.

Evaluating Unlearners

ERASURE comes with the most well-known Unlearning metrics out-of-the-box.

Other metrics, like Accuracy, can be instantiated directly from external libraries, like sci-kit learn.

```
25 "evaluator":{
26  "class": "erasure.evaluations.<NS>.Evaluator",
27  "parameters": { "measures":[ ... ] } }
```

All metrics have access to both the modified model and the original model, to evaluate differences if they are needed.

ERASURE Out-of-the-box

Datasets

- ALL datasets from HuggingFace
- ALL datasets from Torchvision
- ALL datasets from UCI Repository
- ALL datasets from Torch Geometric

Unlearners

- Gold Model
- FineTuning
- Advanced NegGrad
- BadTeaching
- NegGrad
- Eu_k
- SRL
- FisherForgetting
- SalUn
- Composite
- Scrub
- SSD
- Unsir

Evaluations

- ALL metrics from sklearn
- RunTime
- Memory Usage
- UMIA

Extending ERASURE

- Extending Erasure is as simple as creating a class with your logic.
- Demo paper at IJCAI for more info:

- CIKM Resource paper coming soon!

Support ERASURE!

https://github.com/aiim-research/ERASURE

GitHub

ERASURE

PLEASE SUPPORT US WITH A STAR!

Thank you for your attention

UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA

DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

ERASURE

A Machine Unlearning Framework

Flavio Giobergia – *Politecnico di Torino*

Giovanni Stilo – *University of L'Aquila*

3rd European Summer School on Artificial Intelligence

Bratislava, 30/6 – 4/7/2025

ESSAI 2025

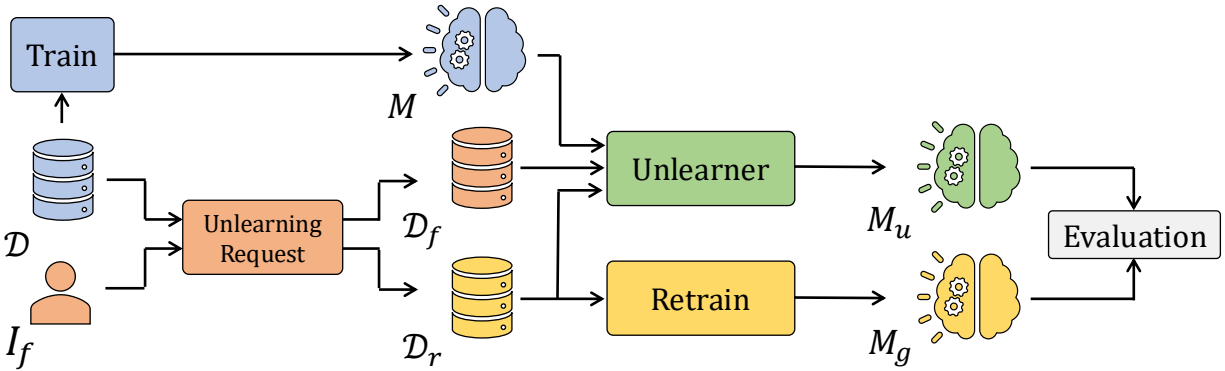
BRATISLAVA - SLOVAKIA

organized by EurAi

Machine Unlearning Workflow

ERASURE is a modular, extensible Machine Unlearning Framework.

ERASURE handles all the steps of a typical Machine Unlearning workflow:



An Unlearning Request is issued by entity I_f, splitting the dataset in Forget Set D_f and Retain Set D_r. The Unlearner method takes them as input, together with the original model M trained on D, and outputs the Unlearned Model M_u. The goal is to make M_u as close as possible to the Gold Model M_g, retrained from scratch on D_r . This closeness is evaluated by several metrics.

What is ERASURE?

https://github.com/aiim-research/ERASURE

GitHub

PLEASE SUPPORT US WITH A STAR!

ERASURE

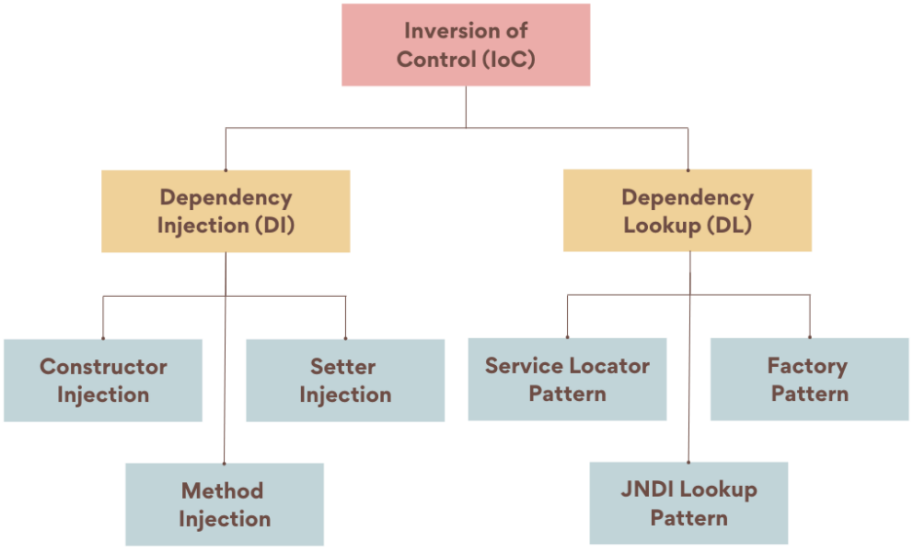
ESSAI 2025

BRATISLAVA - SLOVAKIA

organized by EurAI

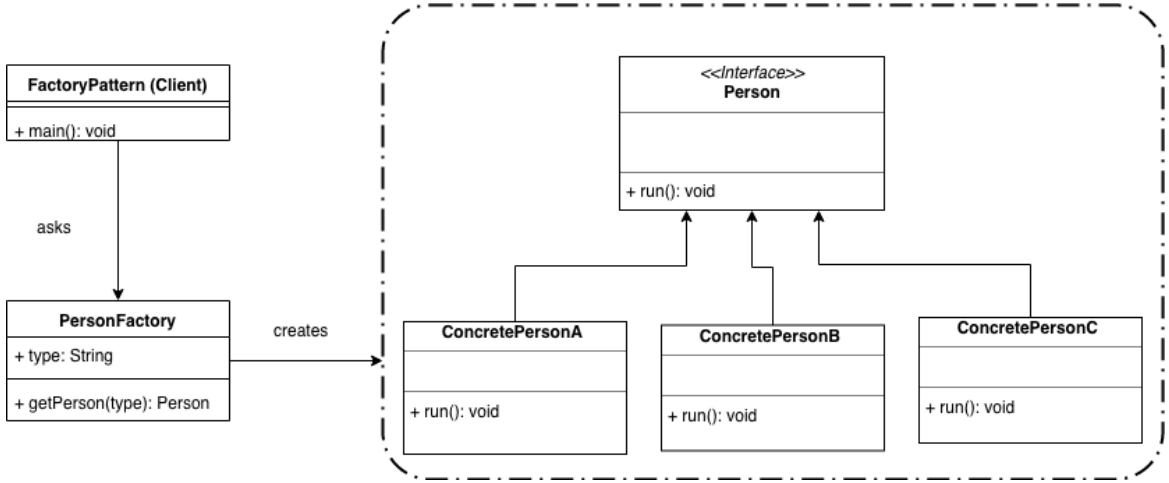
ERASURE is a modular, extensible Machine Unlearning Framework.

Inversion of Control



- **Inversion of Control (IoC):** shifts control of object creation and execution flow from client code to an external framework. IoC reduces tight coupling by delegating dependency management to an external provider, often through *dependency injection*. ERASURE leverages IoC for dynamic dependency management.

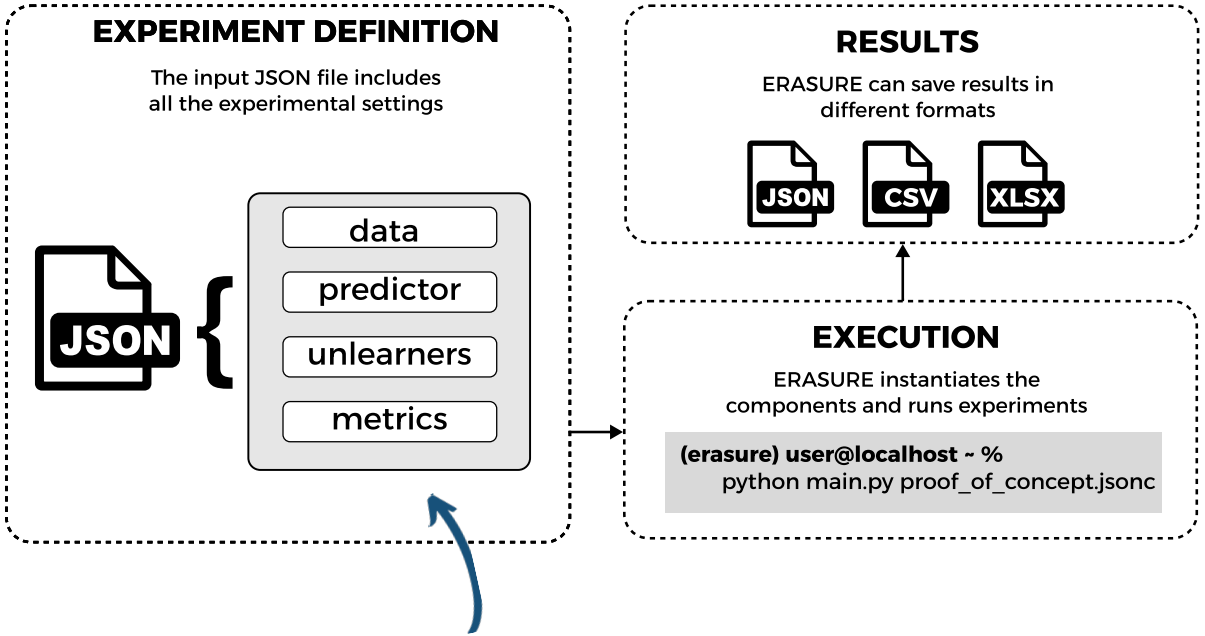
Factory Pattern



- **Factory Pattern**: abstracts object instantiation. It encapsulates object creation within a dedicated factory class, enabling flexible instantiation. ERASURE uses the Factory Pattern to instantiate Datasets, Models, Unlearners, and everything that is needed for a complete Machine Unlearning workflow.

ERASURE's Configuration

ALL the Experimental settings in ERASURE are defined through a single JSON file, called **Main Configuration**.



The Main Configuration file needs to specify all these components.

Main Configuration

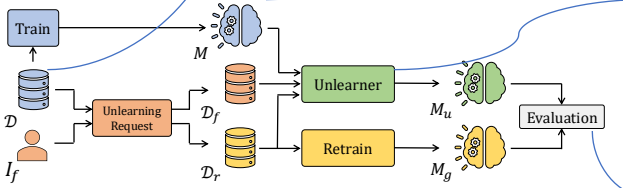
This is the general structure of the Main Configuration. It needs to define four main components:

- **Data**
- **Predictor**
- **Unlearners**
- **Evaluator**

They are shown in red in the Figure.

```
"data": {"class":"erasure.data.<NS>.DatasetManager",
      "parameters": {
         "DataSource": { ... },
         "partitions": [ "p_1", "p_2", ... , "p_n" ]
       }},
"predictor": {"class": "erasure.model.<NS>.TorchModel",
 "parameters": {
   "optimizer": {"class": "torch.optim.Adam"},
   "loss_fn": {"class": "torch.nn.CrossEntropyLoss"},
   "model": {"class": "erasure.<NS>.BERTClassifier"}
 }},
"unlearners":[
   {"compose_gold" : "configs/snippets/u_gold.json"},
   .
   .
   ,
   {"class":"erasure.<NS>.AdvancedNegGrad",
      "parameters": {
           "epochs": 1,
           "ref_data_retain": "retain",
           "ref_data_forget": "forget",
           "optimizer": {"class": "torch.optim.Adam",
                "parameters": {"lr": 0.0001}}}
   }
],
"evaluator":{
 "class": "erasure.evaluations.<NS>.Evaluator",
 "parameters": { "measures":[ ... ] } }
```

Main Configuration (2)




```
"data": {"class":"erasure.data.<NS>.DatasetManager",
      "parameters": {
         "DataSource": { ... },
         "partitions": [ "p_1", "p_2", ... , "p_n" ]
       }},
"predictor": {"class": "erasure.model.<NS>.TorchModel",
 "parameters": {
   "optimizer": {"class": "torch.optim.Adam"},
   "loss_fn": {"class": "torch.nn.CrossEntropyLoss"},
   "model": {"class": "erasure.<NS>.BERTClassifier"}
 }},
"unlearners":[
   {"compose_gold" : "configs/snippets/u_gold.json"},
   .
   .
   ,
   {"class":"erasure.<NS>.AdvancedNegGrad",
      "parameters": {
           "epochs": 1,
           "ref_data_retain": "retain",
           "ref_data_forget": "forget",
           "optimizer": {"class": "torch.optim.Adam",
                "parameters": {"lr": 0.0001}}}
   }
],
"evaluator":{
 "class": "erasure.evaluations.<NS>.Evaluator",
 "parameters": { "measures":[ ... ] } }
```

These are the same modules of the Machine Unlearning workflow.

Main Configuration (3)

The four main components are defined through JSON objects.

Each component is a must define what to instantiate through a class and parameters.

Parameters can themselves be object defined through class and parameters, creating a hierarchical structure in the configuration file.

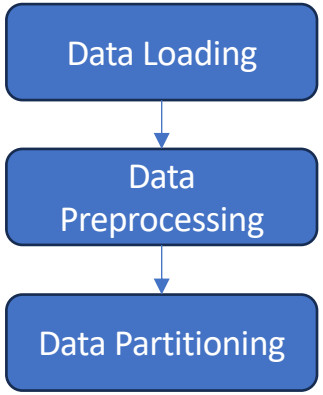
```
"data": {"class":"erasure.data.<NS>.DatasetManager",
      "parameters": {
         "DataSource": { ... },
         "partitions": [ "p_1", "p_2", ... , "p_n" ]
       }},
"predictor": {"class": "erasure.model.<NS>.TorchModel",
 "parameters": {
   "optimizer": {"class": "torch.optim.Adam"},
   "loss_fn": {"class": "torch.nn.CrossEntropyLoss"},
   "model": {"class": "erasure.<NS>.BERTClassifier"}
 }},
"unlearners":[
   {"compose_gold" : "configs/snippets/u_gold.json"},
   .
   .
   ,
   {"class":"erasure.<NS>.AdvancedNegGrad",
      "parameters": {
           "epochs": 1,
           "ref_data_retain": "retain",
           "ref_data_forget": "forget",
           "optimizer": {"class": "torch.optim.Adam",
                "parameters": {"lr": 0.0001}}}
   }
],
"evaluator":{
 "class": "erasure.evaluations.<NS>.Evaluator",
 "parameters": { "measures":[ ... ] } }
```

DatasetManager

```
"data": {"class":"erasure.data.<NS>.DatasetManager",
"parameters": {
   "DataSource": { "class":"erasure.data.<NS>.HFDataSource",
      "parameters": {"path": "user_id/ag_news", ... ,
   "preprocess":[
      {"class":"erasure.data.<NS>.add_z_label.StringContain",
         "parameters":{"contains":["Real Madrid"]}},
      {"class":"erasure.data.<NS>.TokenizeX",
         "parameters":{
            "tokenizer":
            {"class":"erasure.data.<NS>.TokenizerWrapper",
               "parameters":{"tokenizer": "bert-base-uncased",
                     ...}} }} ... ]}},
   "partitions": [
      {"class":"erasure.data.<NS>.DataSplitterByZ",
            "parameters":{"parts_names":["forget", "other"],
            "z_labels":[1]}},
      {"class":"erasure.data.<NS>.DataSplitterPercentage",
            "parameters":{"parts_names":["retain", "test"],
            "percentage":0.8, "ref_data":"other"}},
      {"class":"erasure.data.<NS>.DataSplitterConcat",
            "parameters":{"parts_names":["train","-"],
            "concat_splits":["retain","forget"]}}
      ],
   "batch_size":64}}
```

The Data JSON object always instantiates a DatasetManager object.

The DatasetManager orchestates everything about Data loading, preprocessing and partitioning.



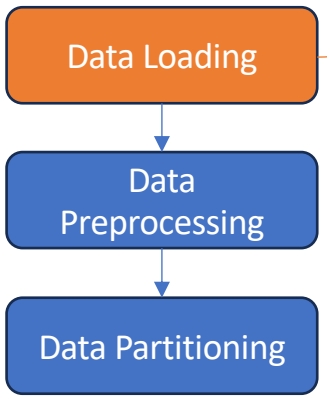
DataSource

```
"data": {"class":"erasure.data.<NS>.DatasetManager",
"parameters": {
   "DataSource": { "class":"erasure.data.<NS>.HFDataSource",
      "parameters": {"path": "user_id/ag_news", ... ,
   "preprocess":[
      {"class":"erasure.data.<NS>.add_z_label.StringContain",
         "parameters":{"contains":["Real Madrid"]}},
      {"class":"erasure.data.<NS>.TokenizeX",
         "parameters":{
            "tokenizer":
            {"class":"erasure.data.<NS>.TokenizerWrapper",
               "parameters":{"tokenizer": "bert-base-uncased",
                     ...}} }} ... ]}},
   "partitions": [
      {"class":"erasure.data.<NS>.DataSplitterByZ",
            "parameters":{"parts_names":["forget", "other"],
            "z_labels":[1]}},
      {"class":"erasure.data.<NS>.DataSplitterPercentage",
            "parameters":{"parts_names":["retain", "test"],
            "percentage":0.8, "ref_data":"other"}},
      {"class":"erasure.data.<NS>.DataSplitterConcat",
            "parameters":{"parts_names":["train","-"],
            "concat_splits":["retain","forget"]}}
      ],
   "batch_size":64}}
```

The DataSource handles Data Loading. It instantiates an object that extends DataSource, based on the source to download the data from.

For instance, HFDataSource handles download of data from HuggingFace.
The parameters passed will be fed to the DataSource directly and can be used to select which dataset to download (in the example, ag_news).

DataSource (2)



The **DataSource** oversees Data Loading. Each source is implemented by a specific DataLoader (e.g., HFDataSource for HuggingFace).

The DataSource will return a DatasetWrapper that must be iterable like Pytorch's DataLoaders.

An out-of-the-box, simple DatasetWrapper is already available in ERASURE.

DataSource (3)

Example implementation of HFDataSource (extending DataSource).

The class must implement __init__, get_name, and create_data.

```python
class HFDataSource(DataSource):
    def __init__(self, global_ctx: Global, local_ctx: Local):
        super().__init__(global_ctx, local_ctx)
        self.dataset = None
        self.path = self.local_config['parameters']['path']
        self.configuration = self.local_config.get("configuration","")
        self.label = self.local_config['parameters']['label']
        self.data_columns = self.local_config['parameters']['data_columns']
        self.to_encode = self.local_config['parameters']['to_encode']
        self.classes = self.local_config['parameters']['classes']

    def get_name(self):
        return self.path.split("/")[-1]

    def create_data(self):
```

Create_data must return an instance of DatasetWrapper, either the base class or a custom class (if specific per-batch processing is needed).

Data Processing

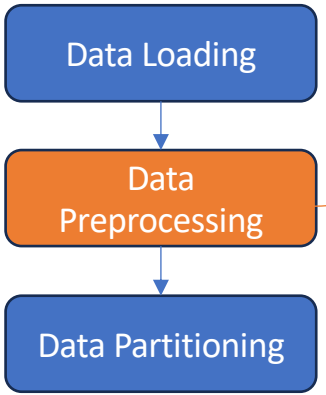
```
"data": {"class":"erasure.data.<NS>.DatasetManager",
"parameters": {
   "DataSource": { "class":"erasure.data.<NS>.HFDataSource",
      "parameters": {"path": "user_id/ag_news", ... ,
   "preprocess":[
      {"class":"erasure.data.<NS>.add_z_label.StringContain",
         "parameters":{"contains":["Real Madrid"]}},
      {"class":"erasure.data.<NS>.TokenizeX",
         "parameters":{
            "tokenizer":
            {"class":"erasure.data.<NS>.TokenizerWrapper",
               "parameters":{"tokenizer": "bert-base-uncased",
                     ...}} }} ... ]}},
   "partitions": [
      {"class":"erasure.data.<NS>.DataSplitterByZ",
            "parameters":{"parts_names":["forget", "other"],
            "z_labels":[1]}},
      {"class":"erasure.data.<NS>.DataSplitterPercentage",
            "parameters":{"parts_names":["retain", "test"],
            "percentage":0.8, "ref_data":"other"}},
      {"class":"erasure.data.<NS>.DataSplitterConcat",
            "parameters":{"parts_names":["train","-"],
            "concat_splits":["retain","forget"]}}
      ],
   "batch_size":64}}
```

DataSource contains a parameter called **«preprocess»,** instantiated by a JSON Array.

Each element in the JSON Array is a subclass of the class Preprocess, which takes X,Y,Z as input and outputs X,Y,Z after a specific preprocess technique is executed.

In this case, the preprocesses instantiated by this JSON file are StringContain and TokenizeX.

Data Processing (1)



The **Preprocess class** oversees Data Preprocessing. Each particular Preprocessing technique is an extension of this class and returns a triple (X,Y,Z).

The parameter Z is useful to filter or further partition the data with more information, which is critical for the task of Machine Unlearning.

Data Processing (2)

Example implementation of StringContain, extending Preprocess.
The class is very simple: the only parameter is 'contains', and it is the substring that is we need to check for membership in each sample of X.

```
class StringContain(Preprocess):
    def __init__(self, global_ctx: Global, local_ctx: Local):
        super().__init__(global_ctx, local_ctx)
        self.contains = self.local_config['parameters']['contains']


    def process(self, X, y, z):

        z = 0
        for string in self.contains:
            if string in X:      You, 4 mesi fa • ag_news …
                z = 1
                break

        return X,y,z
```

If string is contained in the sample (remember that the Preprocess is applied to each batch/each sample), then z is set to 1.

Data Processing (3)

But what's the use of Parameter Z?

ERASURE supports a parameter Z, in addition to the standard X and Y, for filtering. Imagine you want to remove all the images of George Clooney from the dataset CelebA.

There are the labels (Y)

There are the samples (X)

Data Processing (4)

CelebA Backup url - https://shorturl.at/312Yv

There is no information on what celebrity is pictured in each photo! If we only pass the plain dataset (X and Y), we will not be able to filter out pictures, which is critical in Machine Unlearning.

Thanks to the introduction of a Parameter Z, we can pass more information to the classes that will split and filter the data.

Data Processing (5)

```
"data": {"class":"erasure.data.<NS>.DatasetManager",
"parameters": {
   "DataSource": { "class":"erasure.data.<NS>.HFDataSource",
      "parameters": {"path": "user_id/ag_news", ... ,
   "preprocess":[
      {"class":"erasure.data.<NS>.add_z_label.StringContain",
         "parameters":{"contains":["Real Madrid"]}},
      {"class":"erasure.data.<NS>.TokenizeX",
         "parameters":{
            "tokenizer":
            {"class":"erasure.data.<NS>.TokenizerWrapper",
               "parameters":{"tokenizer": "bert-base-uncased",
                     ...}} }} ... ]}},
   "partitions": [
      {"class":"erasure.data.<NS>.DataSplitterByZ",
            "parameters":{"parts_names":["forget", "other"],
            "z_labels":[1]}},
      {"class":"erasure.data.<NS>.DataSplitterPercentage",
            "parameters":{"parts_names":["retain", "test"],
            "percentage":0.8, "ref_data":"other"}},
      {"class":"erasure.data.<NS>.DataSplitterConcat",
            "parameters":{"parts_names":["train","-"],
            "concat_splits":["retain","forget"]}}
      ],
   "batch_size":64}}
```

With this in mind, intuitively, the StringContain preprocess basically sets z=1 for each sample containing the words «Real Madrid».

The second preprocess, TokenizeX, uses a tokenizer from the HuggingFace library to tokenize the text in the X.

Data Partitioning

```
"data": {"class":"erasure.data.<NS>.DatasetManager",
"parameters": {
   "DataSource": { "class":"erasure.data.<NS>.HFDataSource",
      "parameters": {"path": "user_id/ag_news", ... ,
   "preprocess":[
      {"class":"erasure.data.<NS>.add_z_label.StringContain",
         "parameters":{"contains":["Real Madrid"]}},
      {"class":"erasure.data.<NS>.TokenizeX",
         "parameters":{
            "tokenizer":
            {"class":"erasure.data.<NS>.TokenizerWrapper",
               "parameters":{"tokenizer": "bert-base-uncased",
                  ...}} }} ... ]}},
   "partitions": [
      {"class":"erasure.data.<NS>.DataSplitterByZ",
            "parameters":{"parts_names":["forget", "other"],
            "z_labels":[1]}},
      {"class":"erasure.data.<NS>.DataSplitterPercentage",
            "parameters":{"parts_names":["retain", "test"],
            "percentage":0.8, "ref_data":"other"}},
      {"class":"erasure.data.<NS>.DataSplitterConcat",
            "parameters":{"parts_names":["train","-"],
            "concat_splits":["retain","forget"]}}
      ],
   "batch_size":64}}
```

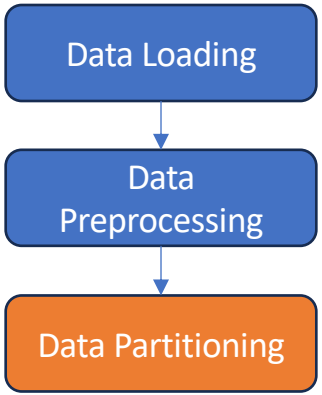
DatasetManager contains a parameter called «partitions», instantiated by a JSON Array.

Each element in the JSON Array is a subclass of the class DataSplitter.

Each DataSplitter splits the data in two, called «parts_names», and optionally accepts a parameter «ref_data» as input.

Data Partitioning (2)

Machine Unlearning requires more demanding Data handling with respect to usual downstream tasks.



Partitioning the data is critical for MU, as it often requires more sophisticated partitioning than conventional train/test splits. For example, most Unlearners rely on Forget and Retain Sets.

To accommodate this, ERASURE introduces a flexible DataSplitter strategy, enabling datasets to be partitioned by configuration in virtually unlimited ways.

Each DataSplitter defines a specific partitioning logic, such as sampling a fixed percentage of data or selecting samples from a given class.

Since they are executed sequentially, the DataSplitters can reference previously created partitions.

Data Partitioning (3)

```
"data": {"class":"erasure.data.<NS>.DatasetManager",
"parameters": {
   "DataSource": { "class":"erasure.data.<NS>.HFDataSource",
      "parameters": {"path": "user_id/ag_news", ... ,
   "preprocess":[
      {"class":"erasure.data.<NS>.add_z_label.StringContain",
         "parameters":{"contains":["Real Madrid"]}},
      {"class":"erasure.data.<NS>.TokenizeX",
         "parameters":{
            "tokenizer":
            {"class":"erasure.data.<NS>.TokenizerWrapper",
               "parameters":{"tokenizer": "bert-base-uncased",
                     ...}} }} ... ]}},
   "partitions": [
      {"class":"erasure.data.<NS>.DataSplitterByZ",
            "parameters":{"parts_names":["forget", "other"],
            "z_labels":[1]}},
      {"class":"erasure.data.<NS>.DataSplitterPercentage",
            "parameters":{"parts_names":["retain", "test"],
            "percentage":0.8, "ref_data":"other"}},
      {"class":"erasure.data.<NS>.DataSplitterConcat",
            "parameters":{"parts_names":["train","-"],
            "concat_splits":["retain","forget"]}}
      ],
   "batch_size":64}}
```

For instance, the first DataSplitter, called DataSplitterByZ, splits the data in »forget» and «other» based on the passed z_labels, in this case [1].

Intuitively, if the sample has z=1, the sample is put in the forget set.

Together with the preprocessing example we saw, we managed to simply and effectively put all the strings containing »Real Madrid» in the Forget Set!

Data Partitioning (4)

```
"data": {"class":"erasure.data.<NS>.DatasetManager",
"parameters": {
   "DataSource": { "class":"erasure.data.<NS>.HFDataSource",
      "parameters": {"path": "user_id/ag_news", ... ,
   "preprocess":[
      {"class":"erasure.data.<NS>.add_z_label.StringContain",
         "parameters":{"contains":["Real Madrid"]}},
      {"class":"erasure.data.<NS>.TokenizeX",
         "parameters":{
            "tokenizer":
            {"class":"erasure.data.<NS>.TokenizerWrapper",
               "parameters":{"tokenizer": "bert-base-uncased",
                     ...}} }} ... ]}},
   "partitions": [
      {"class":"erasure.data.<NS>.DataSplitterByZ",
            "parameters":{"parts_names":["forget", "other"],
            "z_labels":[1]}},
      {"class":"erasure.data.<NS>.DataSplitterPercentage",
            "parameters":{"parts_names":["retain", "test"],
            "percentage":0.8, "ref_data":"other"}},
      {"class":"erasure.data.<NS>.DataSplitterConcat",
            "parameters":{"parts_names":["train","-"],
            "concat_splits":["retain","forget"]}}
      ],
   "batch_size":64}}
```

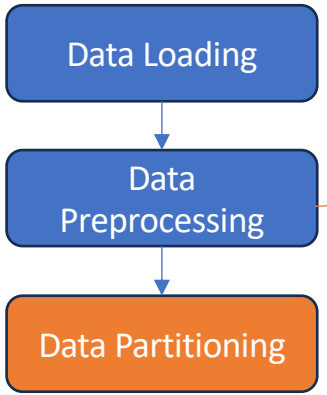
DataSplitter work sequentially, so they can reference previously created partitions.

In this case, DataSplitterPercentage works on the «other» partition, so anything that is not in the Forget Set (as defined by the previous splitter)

➔ Anything that does not contain «Real Madrid»!

Each DataSplitter implements a splitting logic. DataSplitterPercentage just splits based on the given percentage.

Data Partitioning (5)



The **Data Partitioning** is particularly critical for Machine Unlearning. It is implemented by cascading DataSplitters who can reference previously created partitions.

Each DataSplitter references a specific partitions (or the entire dataset when not specified) and implements a particular splitting logic (based on Z, percentage, number of samples, etc…)

Data Example

Example code to load a custom dataset:

```
from erasure.data.datasource import DataSource
from erasure.data.datasets.Dataset import DatasetWrapper

class MyDataSource(DataSource):
    # class initialization omitted

    def get_name(self):
        return self.name

    def create_data(self):
        data = ##data loading logic
        dataset = MyDatasetWrapper(data)
        return dataset

    def get_simple_wrapper(self,data):
        return MyDatasetWrapper(data)


class MyDatasetWrapper(DatasetWrapper):
    # class initialization omitted

    def __realgetitem__(self, index:int):
        X,y = ##sample selection logic
        return X,y

    def get_n_classes(self):
        return len(self.classes)
```

Predictor

We saw the Data section of an ERASURE Main Configuration.

Now we move to the Predictor.

The predictor contains the model that will be trained (and then unlearned).

All models in Pytorch are instantiated via TorchModel.

```
"data": {"class":"erasure.data.<NS>.DatasetManager",
      "parameters": {
         "DataSource": { ... },
         "partitions": [ "p_1", "p_2", ... , "p_n" ]
      }},
"predictor": {"class": "erasure.model.<NS>.TorchModel",
  "parameters": {
    "optimizer": {"class": "torch.optim.Adam"},
    "loss_fn": {"class": "torch.nn.CrossEntropyLoss"},
    "model": {"class": "erasure.<NS>.BERTClassifier"}
  }},
"unlearners":[
    {"compose_gold" : "configs/snippets/u_gold.json"},
    .
    .
    ,
    {"class":"erasure.<NS>.AdvancedNegGrad",
       "parameters": {
            "epochs": 1,
            "ref_data_retain": "retain",
            "ref_data_forget": "forget",
            "optimizer": {"class": "torch.optim.Adam",
                 "parameters": {"lr": 0.0001}}}
    }
],
"evaluator":{
  "class": "erasure.evaluations.<NS>.Evaluator",
  "parameters": { "measures":[ ... ] } }
```

Predictor (2)

Be careful: the Predictor **contains** the model, along with several other parameters that are needed for training, like optimizer, loss function, and epochs.

Predictor and predictor.model are two different entities. The latter is simply the architecture of the model that will be trained.

```
"data": {"class":"erasure.data.<NS>.DatasetManager",
      "parameters": {
         "DataSource": { ... },
         "partitions": [ "p_1", "p_2", ... , "p_n" ]
      }},
"predictor": {"class": "erasure.model.<NS>.TorchModel",
  "parameters": {
    "optimizer": {"class": "torch.optim.Adam"},
    "loss_fn": {"class": "torch.nn.CrossEntropyLoss"},
    "model": {"class": "erasure.<NS>.BERTClassifier"}
  }},
"unlearners":[
    {"compose_gold" : "configs/snippets/u_gold.json"},
    .
    .
    ,
    {"class":"erasure.<NS>.AdvancedNegGrad",
       "parameters": {
            "epochs": 1,
            "ref_data_retain": "retain",
            "ref_data_forget": "forget",
            "optimizer": {"class": "torch.optim.Adam",
                 "parameters": {"lr": 0.0001}}}
    }
],
"evaluator":{
  "class": "erasure.evaluations.<NS>.Evaluator",
  "parameters": { "measures":[ ... ] } }
```

Predictor (3)

For instance, this is the predictor.model that we are instantiating in that config file, a BERTClassifier:

```
class BERTClassifier(nn.Module):
    def __init__(self, n_classes=2):
        super(BERTClassifier, self).__init__()

        self.bert = BertModel.from_pretrained("bert-base-uncased") # automodel

        self.fc1 = nn.Linear(self.bert.config.hidden_size, 512)
        self.fc2 = nn.Linear(512, n_classes)

        self.relu = nn.ReLU()
        self.last_layer = self.fc2

    def forward(self, X):

        input_ids = X[:, 0, :].long()
        attention_mask = X[:, 1, :].long()


        outputs = self.bert(input_ids=input_ids, attention_mask=attention_mask)

        x = outputs.last_hidden_state[:, 0, :]
        x = self.relu(self.fc1(x))

        intermediate_output = x
        x = self.fc2(x)

        return intermediate_output, x
```

Predictor (4)

TorchModel handles all the parameters required for training.
The model is simply the model architecture.

Once all the parameters have been set, TorchModel automatically trains the model according to the given parameters, Optimizer (which can itself be a configurable class), learning rate, epochs, etc.

Training is robust once the Dataset is loaded because DatasetWrapper is required to be iterable like a TorchLoader, thus adapting to batch-level training.

Unlearners

The third section of a JSON Main Configuration file is the “unlearners” JSON Array.

Each element in this JSON Array will instantiate an unlearner to be evaluated.

All unlearners will follow the same class/parameters schema that we have seen so far.

```
"data": {"class":"erasure.data.<NS>.DatasetManager",
      "parameters": {
         "DataSource": { ... },
         "partitions": [ "p_1", "p_2", ... , "p_n" ]
       }},
"predictor": {"class": "erasure.model.<NS>.TorchModel",
 "parameters": {
   "optimizer": {"class": "torch.optim.Adam"},
   "loss_fn": {"class": "torch.nn.CrossEntropyLoss"},
   "model": {"class": "erasure.<NS>.BERTClassifier"}
 }},
"unlearners":[
   {"compose_gold" : "configs/snippets/u_gold.json"},
   .
   .
   ,
   {"class":"erasure.<NS>.AdvancedNegGrad",
      "parameters": {
           "epochs": 1,
           "ref_data_retain": "retain",
           "ref_data_forget": "forget",
           "optimizer": {"class": "torch.optim.Adam",
                "parameters": {"lr": 0.0001}}}
   }
],
"evaluator":{
 "class": "erasure.evaluations.<NS>.Evaluator",
 "parameters": { "measures":[ ... ] } }
```

Unlearners (2)

Consider, for instance, the Unlearner called AdvancedNegGrad.

```
"data": {"class":"erasure.data.<NS>.DatasetManager",
      "parameters": {
         "DataSource": { ... },
         "partitions": [ "p_1", "p_2", ... , "p_n" ]
       }},
"predictor": {"class": "erasure.model.<NS>.TorchModel",
 "parameters": {
   "optimizer": {"class": "torch.optim.Adam"},
   "loss_fn": {"class": "torch.nn.CrossEntropyLoss"},
   "model": {"class": "erasure.<NS>.BERTClassifier"}
 }},
"unlearners":[
   {"compose_gold" : "configs/snippets/u_gold.json"},
   .
   .
   ,
   {"class":"erasure.<NS>.AdvancedNegGrad",
      "parameters": {
           "epochs": 1,
           "ref_data_retain": "retain",
           "ref_data_forget": "forget",
           "optimizer": {"class": "torch.optim.Adam",
                "parameters": {"lr": 0.0001}}}
   }
],
"evaluator":{
 "class": "erasure.evaluations.<NS>.Evaluator",
 "parameters": { "measures":[ ... ] } }
```

It requires a series of parameters, like epochs or optimizer, that can be passed directly from the JSON Configuration file.

Unlearners (3)

To implement a **Custom Unlearner**, one can extend the **TorchUnlearner** class and implement its __unlearn__(self) method.

It's that simple.

- __unlearn__(self) takes nothing as input but has **access to all the parameters** that were passed through the configuration file, all the dataset and datasets partition, and the original model.

- **The original model is given in isolation**, meaning that all Unlearners has access to their copy. So, the modifications made by one Unlearner will not propagate to other Unlearners.

- The __unlearn__(self) method must **return a modified version of the original model**. This returned model will be evaluated lately.

Unlearners (4)

The __init__ method of an Unlearner handles all the parameters, instantiations and things needed for the Unlearner to run.

The method __unlearn__(self) encapsulates the Unlearn logic of the Unlearner.

Implementing a custom Unlearner in ERASURE is as easy as implementing the __unlearn__(self) method with access to all partitions in the dataset and the original model.

ERASURE then handles all the training and evaluation on its own.

Each Unlearner will return a modified version of the original Model.

Unlearners (5)

Example of a custom Unlearner:

```
from erasure.unlearners.torchunlearner import TorchUnlearner

class MyUnlearner(TorchUnlearner):
    def init(self):
        # class initialization

    def __unlearn__(self):
        #The model is already in self
        model = self.predictor.model

        #Get the DataLoaders for train and forget splits
        train_loader, _ = self.dataset.get_loader_for(self.
              ref_data_train, None)

        forget_loader, _ = self.dataset.get_loader_for(self.
              ref_data_forget, None)



        ## Unlearner logic

        return self.predictor
```

Evaluator

Lastly, the Evaluator module handles the evaluation of the models returned from the Unlearners.

The Evaluator is instantiated here, and the parameters contain all the measures that need to be evaluated.

```
"data": {"class":"erasure.data.<NS>.DatasetManager",
      "parameters": {
         "DataSource": { ... },
         "partitions": [ "p_1", "p_2", ... , "p_n" ]
       }},
"predictor": {"class": "erasure.model.<NS>.TorchModel",
 "parameters": {
   "optimizer": {"class": "torch.optim.Adam"},
   "loss_fn": {"class": "torch.nn.CrossEntropyLoss"},
   "model": {"class": "erasure.<NS>.BERTClassifier"}
 }},
"unlearners":[
   {"compose_gold" : "configs/snippets/u_gold.json"},
   .
   .
   ,
   {"class":"erasure.<NS>.AdvancedNegGrad",
      "parameters": {
           "epochs": 1,
           "ref_data_retain": "retain",
           "ref_data_forget": "forget",
           "optimizer": {"class": "torch.optim.Adam",
                "parameters": {"lr": 0.0001}}}
   }
],
"evaluator":{
  "class": "erasure.evaluations.<NS>.Evaluator",
  "parameters": { "measures":[ ... ] } }
```

Evaluator (2)

In ERASURE, the **Evaluator** orchestrates the metrics to be measured for each Unlearner. The **Evaluation** object contains the metrics themselves and is then used to save the output.
The **Measure** objects implement the actual metrics.

ERASURE comes with the most well-known Unlearning metrics out-of-the-box.

Other metrics, like Accuracy, can be instantiated directly from external libraries, like sci-kit learn.

Remember: you're always evaluating a model!

All metrics have access to both the modified model and the original model, to evaluate differences if they are needed.

Evaluator (3)

Example of a custom Measure:

```
from erasure.core.measure import Measure
from erasure.evaluations.evaluation import Evaluation

class MyMeasure(Measure):
    # class inizialization omitted

    def process(self, e: Evaluation):
        original_model = e.predictor
        unlearned_model = e.unlearned_model

        result = self.my_measure_implementation(unlearned_model,
             original_model)

        e.add_value("MyMeasure", result)

        return e

    def my_measure_implementation(self, unlearned_model,
          original_model):
        # implement your logic here
        return measure
```

Support ERASURE

https://github.com/aiim-research/ERASURE

GitHub

PLEASE SUPPORT US WITH A STAR!

ERASURE
ESSAI 2025
BRATISLAVA - SLOVAKIA
organized by EurAi

ERASURE is a modular, extensible Machine Unlearning Framework.