Ferdelance

IDSIA

Mar 25, 2024

QUICK START

Ferdelance is a **distributed framework** intended to be used both as a workbench to develop new distributed algorithm within a Federated Learning based environment, and perform distributed statistical analysis on private data.

Federated Learning is a Machine Learning approach that allows for training models across decentralized devices or servers while keeping the data localized, increasing the privacy of data holders. Instead of collecting data from various sources and centralizing it in one location for training, federated learning enables model training directly on the devices where the data resides. In federated learning the training of models is distributed across a series of data holders (client nodes) that have direct and exclusive access to their data. The particularity of this approach is that the training data never leave these nodes, while only aggregated data, such as model parameters, are exchanged to build an aggregated model.

The current implementation support both a centralized setup, where model's parameters are sent from the client nodes to an aggregation node, and distributed setup, where a model is shared across multiple nodes and multiple model aggregation can happen on different nodes.

The intent of this framework is to develop a solution that enable researcher to develop and test new machine learning models in a federated learning context without interacting directly with the data. The framework wraps a familiar set of Python packages and libraries, such as Scikit-Learn and Pandas. This allows researchers to quickly setup data extraction pipeline, following the *Extract-Transform-Load* paradigm, and build models or analyze data.

CHAPTER

ONE

GETTING STARTED

1.1 Requirements

Ferdelance is a library written with `Python 3.10`.

1.2 Installation

To install a node or a workbench, just install the whole library:

```
pip install ferdelance
```

1.3 Example

```
# %%
from ferdelance.core.distributions import Collect
from ferdelance.core.estimators import GroupCountEstimator, MeanEstimator
from ferdelance.core.model_operations import Aggregation, Train, TrainTest
from ferdelance.core.models import FederatedRandomForestClassifier,␣
↪StrategyRandomForestClassifier
from ferdelance.core.steps import Finalize, Parallel
from ferdelance.core.transformers import FederatedSplitter, FederatedKBinsDiscretizer
from ferdelance.schemas.workbench import WorkbenchResource
from ferdelance.workbench import (
    Context,
    Project,
    Client,
    Artifact,
    ArtifactStatus,
    DataSource,
)

import numpy as np

import json
```

(continues on next page)

(continued from previous page)

```
# %% create the context
ctx = Context("http://localhost:1456")

# %% load a project given a token
project_token = "58981bcbab77ef4b8e01207134c38873e0936a9ab88cd76b243a2e2c85390b94"

project: Project = ctx.project(project_token)

# %% What is this project?

print(project)

# %% (for DEBUG) ask the context for clients of a project
clients: list[Client] = ctx.clients(project)

for c in clients:
    print(c)

# %% (for DEBUG) ask the context for data source of a project
datasources: list[DataSource] = ctx.datasources(project)

for datasource in datasources:
    print(datasource)  # <--- non aggregated

# %% working with data

ds = project.data  # <--- AggregatedDataSource

print(ds)

# this is like a describe, but per single feature
for feature in ds.features:
    print(feature)

# %% develop a filter query

# prepare transformation query with all features
q = project.extract()

# inspect a feature data type
feature = q["variety"]

print(feature)

# add filter to the extraction query
q = q.add(q["variety"] < 2)

# %% add transformer

q = q.add(
    FederatedKBinsDiscretizer(
        features_in=[q["variety"]],
```

(continues on next page)

(continued from previous page)

```
        features_out=[q["variety_discr"]],
    )
)

# %% statistics 1

gc = GroupCountEstimator(
    query=q,
    by=["variety_discr"],
    features=["variety_discr"],
)

ret = ctx.submit(project, gc.get_steps())

# %% statistics 2

me = MeanEstimator(query=q)

ret = ctx.submit(project, me.get_steps())

# %% prepare the model steps
model = FederatedRandomForestClassifier(
    n_estimators=10,
    strategy=StrategyRandomForestClassifier.MERGE,
)

label = "MedHouseValDiscrete"

steps = [
    Parallel(
        TrainTest(
            query=project.extract().add(
                FederatedSplitter(
                    random_state=42,
                    test_percentage=0.2,
                    label=label,
                )
            ),
            trainer=Train(model=model),
            model=model,
        ),
        Collect(),
    ),
    Finalize(
        Aggregation(model=model),
    ),
]

# %% submit the task to the node, it will be converted to an Artifact

a: Artifact = ctx.submit(project, steps)
```

(continues on next page)

(continued from previous page)

```
print(json.dumps(a.model_dump(), indent=True))  # view execution plan

# %% monitor learning progress
status: ArtifactStatus = ctx.status(a)

print(status)

# %% list produced resources
resources: list[WorkbenchResource] = ctx.list_resources(a)

for r in resources:
    print(r.resource_id, r.creation_time, r.is_ready)

# %% get latest produced resource (the result)

agg_model: FederatedRandomForestClassifier = ctx.get_latest_resource(a)["model"]

# %%

print(agg_model.predict(np.array([[0, 0, 0, 0]])))
print(agg_model.predict(np.array([[1, 1, 1, 1]])))
```

CHAPTER

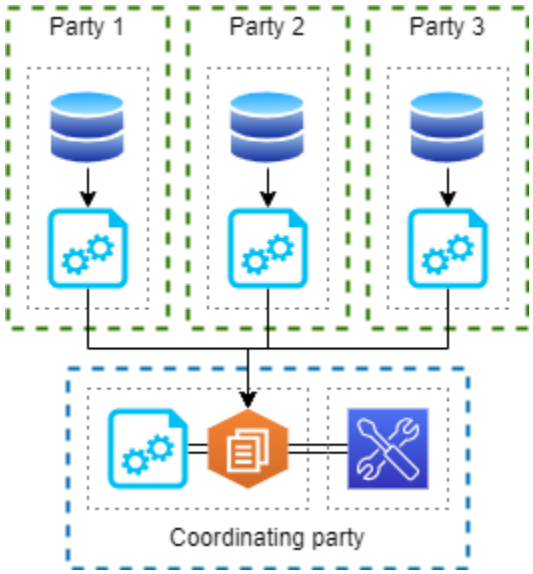
TWO

DISTRIBUTION TOPOLOGY

A topology is how the nodes are distributed across the network. This can influence the types of algorithms and Artifact that can be submitted to the network.

Based on the configuration of each node, the framework can work in two topologies: *centralized* and *decentralized*.

2.1 Centralized topology



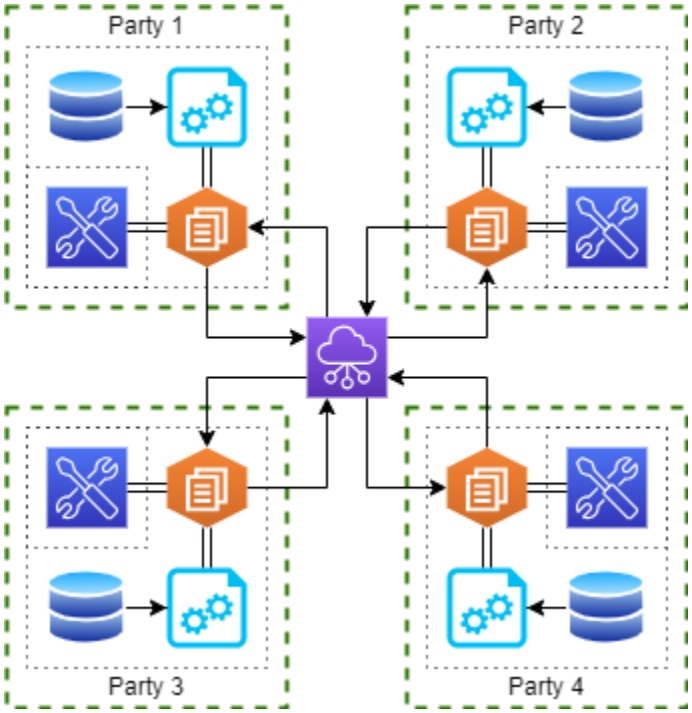
In a *centralized topology* a central node act both as an aggregator and a coordinator for all the nodes in the network.

This is the only available topology for worker nodes in **client** mode, although nodes in **node** mode can still join such network.

_Workbenches* can both be internal o external respect to the consortium network, but the need to connect to the aggregator node to submit the Artifacts.

Sometimes, in this topology, the aggregator node does not share local data and work with the resources produced by the client nodes.

2.2 Decentralized topology



In a *decentralized topology* all nodes can connect and communicate with all the other nodes part of the network.

In this situation, the *workbenches* can connect to any node. When a new Artifact is deployed, the receiver will became the scheduler and the aggregation node for the specified Artifact.

This topology allows a more fluid exchange of resources; given the fact that a nodes knows each other, the aggregation and fusion algorithms can be more complex and leverage on the structure of the network itself to operate.

2.3 Hybrid topology

We have an *hybrid* topology when we mix nodes in *client* mode with other in *node* mode. In this situation, clients can still participate in the creation of resources but they need to send resources through a proxy node. This limits the available types of algorithms and at the same time slows the performance of the network, given the fact that we need to use proxy nodes.

CHAPTER

THREE

SECURITY AND IDENTIFICATION

3.1 Data encryption

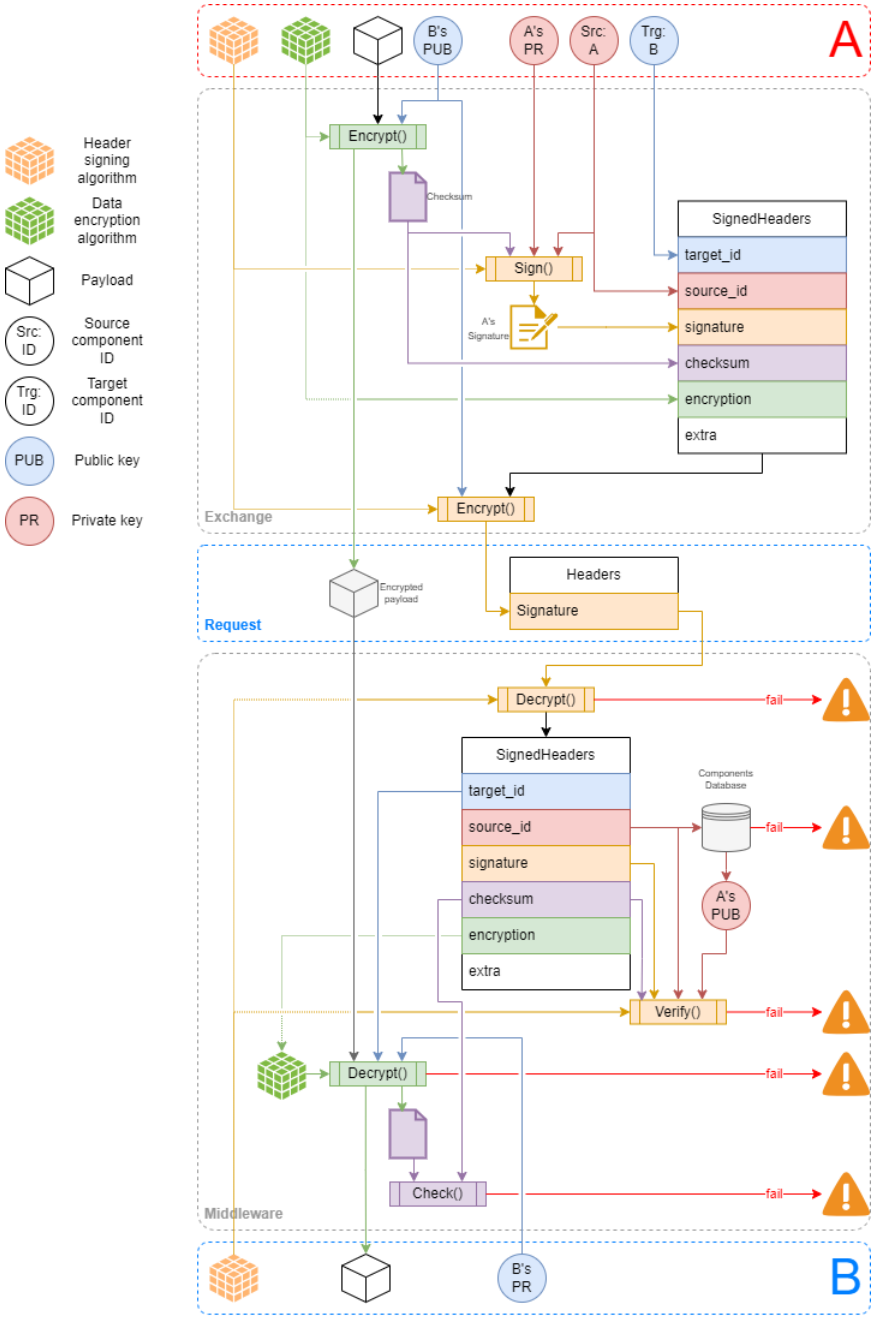
Exchange of data between two nodes is encrypted. In this case, *data* is not limited to resources and model parameters but refers also to all the communication and exchange of metadata that happens between two nodes.

Encryption is done using two algorithms. The first is an encryption algorithm with asymmetric keys. Each node, included the workbench, at startup search for a private key that will be used to sign and encrypt data. The signature of data is used to ensure that the author of a data payload has been sent by the real author of the content, while the encryption algorithm make sure that only the real receiver of a package can read the data.

The second encryption algorithm is used when there is the need to exchange a substantial quantity of data. This algorithm still uses the asymmetric algorithm to encrypt a symmetric key that will be used to encrypt the data. In this ways there is no limit to the amount of data that can be exchanged between two nodes.

When a new node connect to the network for the first time, it shares its public key to all other component of the network. To join a network, a node requires the base url of a join node. Each node offers the `/key` endpoint where the public key is freely available.

3.2 Node Identification



The above image show how the Identification of the author and the security mechanism works. Note that the exchange between node `A` and `B` works in both direction: not only the requests are encrypted and checked using this workflow, but also the responses.

CHAPTER

FOUR

EXECUTION MODE

A node in Ferdelance can be executed in three different flavours, based on the needs of the data provider: `node`, `client`, and `standalone` mode.

4.1 Node mode

This is the standard execution mode of a node. In this mode, the API are accessible from the extern. At least one node in the network need to be executed with this mode to allow other nodes to join.

4.2 Client mode

This is a special mode suited for all the nodes that should not be reachable from the external world. In this mode, after joining a network through a node that will be marked as `scheduler`, the client node will launch a polling thread that will interact with the join node at regular intervals. At each interaction, the node will requires an updates on the task to execute. When the scheduler node notifies a new job to be executed, it is the client node to contact the scheduler and fetch all the required data.

In this mode the APIs are accessibly only locally for maintenance purposes. The other nodes will not be able to interact directly with this node.

Although this guarantees an higher level of security, this mode has the drawback that some algorithms cannot be executed or need to be executed in a different way, such as through the use of the scheduler node as proxy node.

4.3 Standalone mode

One of the simplest way to execute and test changes to the framework is through the so called `standalone mode`. In this mode, the framework is executed as a standalone application. The hardcoded configuration guarantees that a scheduling node capable of scheduling jobs for itself with is deployed. To run in standalone mode there is no need to pass, although it is possible, a configuration parameter:

```
python -m ferdelance.standalone
```

This mode is not suitable for production environment and should be used only for learning of testing purposes.

CHAPTER

FIVE

NODE

The *aggregation node* is a node reachable from all other nodes in the network and the central node of the framework. All workbenches send their payload, called Artifacts, to the aggregation node; while all the clients query the same node for the next job to run. This allows the clients to have more control on the access and an additional layer of protection: a client node is not reachable from the internet and it is the client that contact the known reference node and initiate the execution process.

The node is composed by a web API written with FastAPI that runs and spawns Ray tasks. The node also uses a database to keep track of every stored object.

The easiest way to deploy a node is using **Docker Compose**.

The file docker-compose.integration.yaml contains a definition of all services required to create a stack that simulates a central server node and some client nodes.

5.1 Installation

The installation of a node is simple:

```
pip install ferdelance
```

Once installed it can be run by specifying a YAML configuration file:

```
python -m ferdelance -c ./config.yaml
```

Once one node is up and running, with default parameters the node will be reachable at *http://server:1456/*.

5.2 Node configuration

The minimal content of the configuration file is the definition of the server url to use and at least one datasource. The datasource must have a name and be associated with one or more project thought the `token` field.

```
workdir: ./storage                  # OPTIONAL: local path of the working directory

mode: node                          # one of: node, client, standalone

node:
  name: FerdelanceNode
  healthcheck: 3600.0               # wait in seconds for check self status
  heartbeat: 10.0                   # wait in seconds for clients to fetch updates
```

(continues on next page)

(continued from previous page)

```
  allow_resource_download: true     # if false, nobody can download resources from this␣
↪node

  protocol: http                    # external protocol (http or https)
  interface: 0.0.0.0                # interface to use (0.0.0.0 for node, "localhost"␣
↪for clients)
  url: ""                           # external url that the node will be reachable at
  port: 1456                        # external port to use to reach the APIs

  token_projects_initial:           # initial projects available at node start
    - name: my_beautiful_project    # name of the project
      token: 58981bcbab...          # unique token assigned to the project

join:
  first: true                       # if true, this is the first node in the distributed␣
↪network
  url: ""                           # when a node is note the first, set the url for the␣
↪join node

datasources:                        # list of available datasources
  - name: iris                      # name of the source
    kind: file                      # how the datasource is stored (only 'file')
    type: csv                       # file format supported (only 'csv' or 'tsv')
    path: /data/iris.csv            # path to the file to use
    token:                          # list of project token that can access this␣
↪datasource
    - 58981bcbab7...

database:
  username: ""                      # username used to access the database
  password: ""                      # password used to access the database
  scheme: ferdelance                # specify the name of the database schema to use
  memory: false                     # when set to true, a SQLite in-memory database will␣
↪be used
  dialect: sqlite                   # current accepted dialects are: SQLite and␣
↪Postgresql
  host: ./sqlite.db                 # local path for local file (QSLite) or url for␣
↪remote database
  port: ""                          # port to use to connect to a remote database
```

Note: It is also possible to specify environment variables in the configuration file using the syntax `${ENVIRONMENT_VARIABLE_NAME}` inside the fields of parameters. This is specially useful when setting parameters, such as domains or password, through a Docker compose file.

For the first node of the distributed network, the `join.first` parameter must always be set to `true`. In the network it must always be a first node with this configuration. In all the other cases, both for `client` and `node` mode, the configuration need to specify the `join.url` parameter to a valid url of an existing node. Only urls of nodes in `node` mode can be used in this parameter.

5.3 Database configuration

Database configuration is completely optional. Every node needs a database to work properly. Minimal setup is to use an SQLite in-memory database by setting `database.memory: true`. If not database is configured, then the in-memory database will be used. Other supported database are:

- SQLite file database:

```
database:
  scheme: ferdelance
  dialect: sqlite
  host: ./sqlite.db
  memory: false
```

- Postgresql remote database:

```
database:
  username: "${DATABASE_USER}"
  password: "${DATABASE_PASSWORD}"
  scheme: ferdelance
  dialect: postgresql
  host: remote_url
  port: 5432
  memory: false
```

CHAPTER

SIX

WORKBENCH

The *workbench* is not a standalone application but a library that need to be imported. It is used to communicate with a node and submit Artifacts, that encapsulate instructions for the job scheduling and execution.

Installation is straightforward:

```
pip install ferdelance[workbench]
```

Once installed, just create a context object and obtain a project handler with a token. A project is a collection of data sources. The token is created by the node network administrator and it is unique for each project.

Following an example of how to use the workbench library to connect to a node.

```
from ferdelance.core.distributions import Collect
from ferdelance.core.model_operations import Aggregation, Train, TrainTest
from ferdelance.core.models import FederatedRandomForestClassifier,␣
↪StrategyRandomForestClassifier
from ferdelance.core.steps import Finalize, Parallel
from ferdelance.core.transformers import FederatedSplitter
from ferdelance.workbench import Context, Artifact

server_url = "http://localhost:1456"
project_token = "58981bcbab77ef4b8e01207134c38873e0936a9ab88cd76b243a2e2c85390b94"

# create the context
ctx = Context(server_url)

# load a project
project = ctx.project(project_token)

# an aggregated view on data
ds = project.data

# print all available features
for feature in ds.features:
    print(feature)

# create a query starting from the project's data
q = ds.extract()
q = q.add(q["feature"] < 2)

# create a Federated model
model = FederatedRandomForestClassifier(
```

(continues on next page)

(continued from previous page)

```
    n_estimators=10,
    strategy=StrategyRandomForestClassifier.MERGE,
)

label = "label"

# describe how to distribute the work and how to train teh model
steps = [
    Parallel(
        TrainTest(
            query=project.extract().add(
                FederatedSplitter(
                    random_state=42,
                    test_percentage=0.2,
                    label=label,
                )
            ),
            trainer=Train(model=model),
            model=model,
        ),
        Collect(),
    ),
    Finalize(
        Aggregation(model=model),
    ),
]

# submit Artifact
Artifact: Artifact = ctx.submit(project, steps)
```

CHAPTER

SEVEN

OVERVIEW

The workflow in Ferdelance always starts with a workbench submitting an `Artifact` to a scheduler node. Then, the node will elaborate the Artifact and split it in `tasks` (or `jobs`, in this context these two words are used as synonym) and scheduled based on the worker nodes that will execute each task. The advancement in the completion of the Artifact is strictly controlled by the scheduling node. During the execution of the tasks, the worker nodes will share `resources` between them, as defined in the Artifact. Once all task have been completed, the final resource (it can be a result of a query, or a trained model) is returned to the scheduling node, where it will be possible to download it, if the node configuration allows it.

7.1 Artifact

This is the core unit of the framework. This object defines the sequence of *steps* that will be deployed and execute in the worker network. Workbenches submit Artifacts to a node in charge to act as a scheduler.

Once submitted, an Artifact is converted to a sequence of jobs based on the available worker nodes. The collection of available nodes define the `Scheduler Context`. The scheduler node uses this context and the list of steps defined in the Artifact, to define how the jobs will be executed by which worker. The chosen workers will at this point fetch and execute the tasks assigned.

At any time, the workbench will be able to query the scheduler on the status of the Artifact and following its development. Once all tasks defined by an Artifact are completed, and the scheduler node's configuration allows it, it will be possible to download the produced resources.

7.2 Step

The concept of `Steps` can be explained with an example. Imagine to following this simple algorithm:

```
data = load_data("awesome_data.csv")
data = filter(data.x > 0.5)
res = count(data)
```

This is just a simple algorithm to load some data from disk, select only the rows where `x > 5`, and count them. Now let's make the algorithm distributed, in other words the input data have been split across multiple nodes:

```
nodes = [1,2,3]
res = 0

for n in nodes:
    data = load_data(f"awesome_data_{n}.csv")
```

(continues on next page)

(continued from previous page)

```
    data = filter(data.x > 0.5)
    res += count(data)
```

The result we obtain in `res` variable is the same as before, but whe have to loop over each node to count how many rows we have in total.

A `step` is the code defined inside the loop, with the nuance that it is not executed on the same machine but across multiple machines:

```
# step on node 0 (scheduler)
res_out = 0
send(1, res_out)

# step on node 1
data = load_data("awesome_data_1.csv")
res_out = step.run(res_in, data)
send(2, res_out)

# step on node 2
data = load_data("awesome_data_2.csv")
res_out = step.run(res_in, data)
send(3, res_out)

# step on node 3
data = load_data("awesome_data_3.csv")
res_out = step.run(res_in, data)
send(0, res_out0) # back to the scheduler
```

The step on node 0 will perform an initialization. The step on node 1, 2, and 3 will instead perform the same operations: filtering of the data, count of the remaining rows, and add to the input received from the previous node. The operations of `load_data()` and `send()` are defined by the execution process inside a worker node. This aims to prepare the same *working environment* (mostly a dictionary of variables) on all workers. xIn this way, each step will always have access to the required data, from local source or received as external resources, and will produce a resource.

7.3 Resources and Distribution

A `resource` is anything that can be exchanged between two nodes and that can be consumed or produced by a task. Each task will always produce a resource, also named *a product*, that will be consumed by the next worker node based on the scheduled task. A task can consume a resource produced by a previous node, but there are tasks that will just work with local data that does not consume extra resources.

The flow of *consume a resource, produce a resource* where the type of resource does not change (as an example, a simple count or a mean) is also named an *update of the resource*.

How these resources are practically exchanged between nodes is defined by `Distribution` algorithms. Distribution algorithms need to be defined in an Artifact. Basic distribution algorithm consist in send a resources to all worker nodes and then collect the produced resources in one point. More complex distribution algorithms can have a more fine graded control over this procedure allowing more complex exchanges between nodes.

An example of a more complex algorithm is a *sequential distribution*. In this case, the worker nodes are arranged in a list and their products are sent only to the next node in the list. This distribution continues until all workers have updated the resource. This sequential algorithm is useful when there is the need to update a resource where the merge

of all resources are not possible because, as an example, there are security reasons involved, or the merge is linear and the order is important.

Distribution algorithms are of two types: when instruct a node to send resources to another node we define them as `Distributors`; when it is the opposite, where a node need to search the required resources by previous nodes, we define them as `Collectors`.

CHAPTER

EIGHT

SCHEDULING

Scheduling is the act of convert the steps in an *Artifact* to `SchedulerJob` stored in the *database*. The database contains only metadata regarding the order of the jobs execution. `Step`s, associated with a job, are sent in their entirety to the worker node. This mean that a single *step* is a complete and autonomous piece of data that instruct a node on all the work it needs to do.

8.1 SchedulerJob

A `SchedulerJob` defines the *worker* that will execute the task, a series of *locks*, and the *resources* required and produced.

The *locking mechanism* is the most important part of the whole library since it defines the order of jobs of an Artifact needs to be completed. The list of *locks* in this object is a list of all the jobs that can start once this job is completed successfully. In case of an error in a single job, the whole Artifact fails.

In the database the *locks* are stored in a dedicated table. Once a job is completed, this table is updated. Each time the scheduler is queried for the next available job, the data in the table defines which job to start next.

At higher level, this defines and stores a *Direct Acyclic Graph* (also simply named *DAG*) of the operations.

8.2 Binding

The conversion from sequence of steps to a DAG of operations requires a `SchedulerContext`. This is a description of the available nodes that can execute the tasks. A worker node is valid when it has *data* that are part of the Artifact's project.

A particular worker of the *SchedulerContext* is the `Initiator`. This worker, in the vast majority of the cases, is the scheduler node itself. An *Initiator* does not requires to have local data to be part of the context, since it is used to perform initialization (setup), aggregation, and completion (teardown) operations.

The context keep also track of the numeric ids of the jobs. These ids will be used to keep track of *locks* between jobs during the binding procedure. Once completed, they are converted to UUIDs before storing them in teh database.

The *binding procedure* creates in fact the *locks* between the jobs. It starts from iterating pairwise through the steps, converting each step in a *SchedulerJob*, and considering the distribution operations associated to each step to build the *locks*.

CHAPTER

NINE

CORE OBJECTS

Core objects can be instantiated from class defined in the `ferdelance.core` submodule. In this page we explain the thought flow from a code point of view.

9.1 Entities

Anything that can be exchanged through the API of a node and compose an Artifact is an *entity*. `Entity` is the main class that defines the hierarchy of objects that can be used in an Artifact. This class defines a `class_registry` that is used by the API to convert JSON objects to Python objects, and vice versa, using Pydantic's un/marshalling mechanism.

This is a design choice to make possible the transferring of "code" between a workbench and the worker nodes. The library will provide a series of *blocks of code* ready to be used. The composition of such elements creates an `Artifact`.

New classes must extend the Entity class in order to work.

9.2 Artifacts

`Artifact` class is an entity with a Sequence of *steps*. The Artifact defines how `SchedulerJob`s objects are created given a `SchedulerContext`. The creation is simple: for each pair of consecutive steps, transform the jobs and bind them. Binding means writing a direct acyclic graph defining the Artifact jobs using a lock mechanism. These locks defines which job waits for other jobs to complete successfully before it can be executed.

9.3 Steps

A `Step` is a fundamental abstract class that, in fact, defines an Artifact. Steps have three important methods:

- `jobs()` convert the step in a sequence of `SchedulerJob`s (a single step can generate many jobs).
- `bind()` executed by the scheduler, it is used to create a connection between the step itself, the `SchedulerJob`s before and the `SchedulerJob` after the step execution.
- `step()` executed by the workers, runs the code based on the available local data.

A main implementation of these methods is offered by following classes:

- `BaseStep`, as the name suggests, is a basic implementation of a generic Step and it is composed of an `Operation` (the work to do) object and a `Distribution` object (how to share the resources between nodes).
- `Iterate`, instead, is a *"meta step"* that wraps a sequence of steps allowing the repetition of them, up to a limited number.

The `Iterate` class is needed to allow the scheduler to implement a particular kind of bind that allows the iteration. In fact, this class does not implement a real `bind()` method, instead wraps it in a continued for loop.

The `BaseStep` class is extended by the following classes, allowing the execution of basic procedures and implementing at the same time a distribution:

- `Initialize` schedules a setup operation on the initiator node.
- `Finalize` schedules a teardown operation on the initiator node.
- `Parallel` schedules an operations in parallel across all available worker nodes.
- `Sequential` schedules an operation on each node where the execution of the next job requires the completion of all the previous jobs.
- `RoundRobin` is a particular step where the resources are exchanged following a circle. Let's assume we have `N` workers. Each worker receive a task that updates a resource using local data. Once updated, this resource is sent to the next node that will update it with its local data. This update-and-exchange is iterated until all nodes have contributed to the resource update. In fact the updates happens in parallel: received the initial resource, all nodes update it, and then send it to the next worker, updating. After `N` updates, all nodes have the resources with the contributions of all workers.

9.4 Distributors

Distribution objects defines how the resources are distributed between nodes. The distribution of resources and the *locking* mechanism are tight together. When a job is *locked*, in fact it is waiting for all required resources to be available.

In general, a task can be in charge of both pull the required resources from other nodes or push them to the next node. In both cases, when a step is executed it expects to have all resources available locally, independently if they need to be produced (resources) or not (local data sources).

The main Distribution class is the `Arrange` base algorithm that allows the nodes to collect resources from, and distribute them to multiple nodes. The variant `Distribute` allows the distribution from one node to multiple nodes, while `Collect` awaits for the availability of multiple resources sent to one node. In fact, these two variants are particular cases of the Arrange class.

9.5 Operation and Worker's task

The task that will be executed by a worker node is essentially defined through the `Operation` class.

The first, is that to manipulate and select local data, a `Query` need to be defined and performed. Queries are much like Panda's DataFrame operations, given also the fact that this is the underling library used to manipulate local data.

The second and third concept are interchangeable and cannot coexist at the same time. Each task will produce either a trained `Model` or an `Estimator`. `Model`s starts from the data selected through the Query and are assembled through Machine Learning training algorithms. Once a Model has been built it became a resource that can be exchanged.

An `Estimator`, on the other hand, is much more similar to a results of a database's query. Exploratory analysis, statistical data, and aggregated information can be collected from the distributed data trough the use of a dedicated Estimator.

9.5.1 Queries and Data Transformers

Queries to select, manipulate, and create new features can be created through the composition of `QueryFilters` and the use of `Transformers`.

Each column available in the local data is a `QueryFeature`. From each of them it is possible to concatenate `QueryFilters`, to reduce the number of data available, and create complex `FederatedPipeline` of Transformers to manipulate the data.

These Transformers are wrappers of existing Scikit Learn pre-processing algorithms.

9.5.2 Models and Model Operations

`Model` is an abstract class that offers all the methods to *load()*, *save()*, *train()*, *aggregate()*, and use a Machine Learning Model.

Given a distributed context like in a Federated Learning environment, the *aggregation process* of a Machine Learning model is something that is defined by the model itself.

The *eval()* method implements a generic and powerful analysis of the produced model.

As one can see, the *train()*, *predict()*, and *classify()* methods requires input data in form of X and Y parameters. These data can be created using `ModelOperations`. Through the composition of operations such as `Train`, `TrainTest`, or `LocalCrossValidation`, the worker node will be instructed to perform machine learning task to prepare the data and train the models.

9.5.3 Estimators

The `Estimator` abstract class offers a way to manipulate the output of a `Query` and perform statistical analysis such as:

- `CountEstimator` and `GroupCountEstimator` are used to count the number of records available across the nodes.
- `MeanEstimator` and `GroupMeanEstimator` are used to get the mean of the variable across the nodes.

Count and mean operations uses *noise* and requires a Sequential task order to increase privacy and hide the number of records on each node with data.

The objective of these blocks is to offer to the researchers a way to inspect data in a secure and privacy friendly way, keeping at the same time a familiar way of visualize and interact with the distributed data.

CHAPTER

TEN

SETUP

The Ferdelance framework is open for contributions and offer a quick development environment.

It is useful to use a local Python virtual environment, like virtualenv or conda, during the development of the library.

The repository contains a `Makefile` that can be used to quickly create an environment and install the framework in development mode.

To install the library in development mode, use the following command:

```
pip install -e ".[dev]"
```

To test the changes in development mode there are a number of possibilities:

- standalone mode,
- unit tests using `pytest`,
- integration tests using Docker,
- full development using Docker.

CHAPTER

ELEVEN

TESTING

For testing purposes it is useful to install the test version of the framework:

```
pip install ferdelance[test]
```

11.1 Integration tests

Integration tests are the perfect entrypoint for start deploying and use the framework. These tests simulates a real deployment, although on the same machine, with a dataset split and shared across multiple nodes.

The execution requires a special Docker Compose that will produce a stack with:

- repository with the packed wheel of the library
- a postgres database
- a node acting as an aggregation server
- 2 nodes in client mode
- 2 nodes in default mode (*not used yet*)
- a workbench service

The two client nodes and the two default nodes include the California Housing Pricing dataset. This dataset has been split in three: two parts for the nodes, one part for the evaluation in the workbench. These datasets are saved in CSV format in the data folder.

Configuration of single nodes are stored in the conf folder in YAML format.

Integration tests are written as scripts and simulates what an user could write through the workbench interface. Although a little bit primitive (and not so fast to setup and teardown), it is an effective way to test the workflow of the framework.

All integration tests should be placed in the `tests/integration/tests` folder. These test should be named following the convention `test_NNN.<name>.py`, where `NNN` is an incremental number padded with zeros and `<name>` is just a reference.

To execute the integration tests, simply run the following command from inside the integration folder:

```
make start
```

11.2 Unit tests

To test single part of code, such as transformers, models, or estimators, it is advised to write test files using the pytest library.

A simple test case can be setup as follow:

```
from ferdelance.server.api import api

from fastapi.testclient import TestClient
from sqlalchemy.ext.asyncio import AsyncSession

from tests.utils import connect
import pytest

@pytest.mark.asyncio
async def test_workbench_read_home(session: AsyncSession):
  with TestClient(api) as server:
    args = await connect(server, session)
        wb_exc = args.wb_exc

        res = server.get(
            "/workbench",
            headers=wb_exc.headers(),
        )

        assert res.status_code == 200
```

The fixture to connect to the test db (which for tests is an SQLite database) through the *session* object are defined in the conftest.py file.

Other utility (component connection, clients operations, ...) methods are defined in the `tests/utils.py` file.

> **Warning:** This is an experimental feature, not yet fully implemented or tested.

CHAPTER

TWELVE

PLUGINS

The extension of the library is done through the creation of Python's plugin.

All worker nodes need to have access to the same plugins in order to perform the required tasks. Although technically doable, the installation of required plugins is not done by a running node.

CHAPTER

THIRTEEN

CONTACT AND SUPPORT

Ferdelance has been developed at the Swiss AI Lab IDSIA (Istituto Dalle Molle di Studi sull'Intelligenza Artificiale), a not-for-profit research institute for Artificial Intelligence.