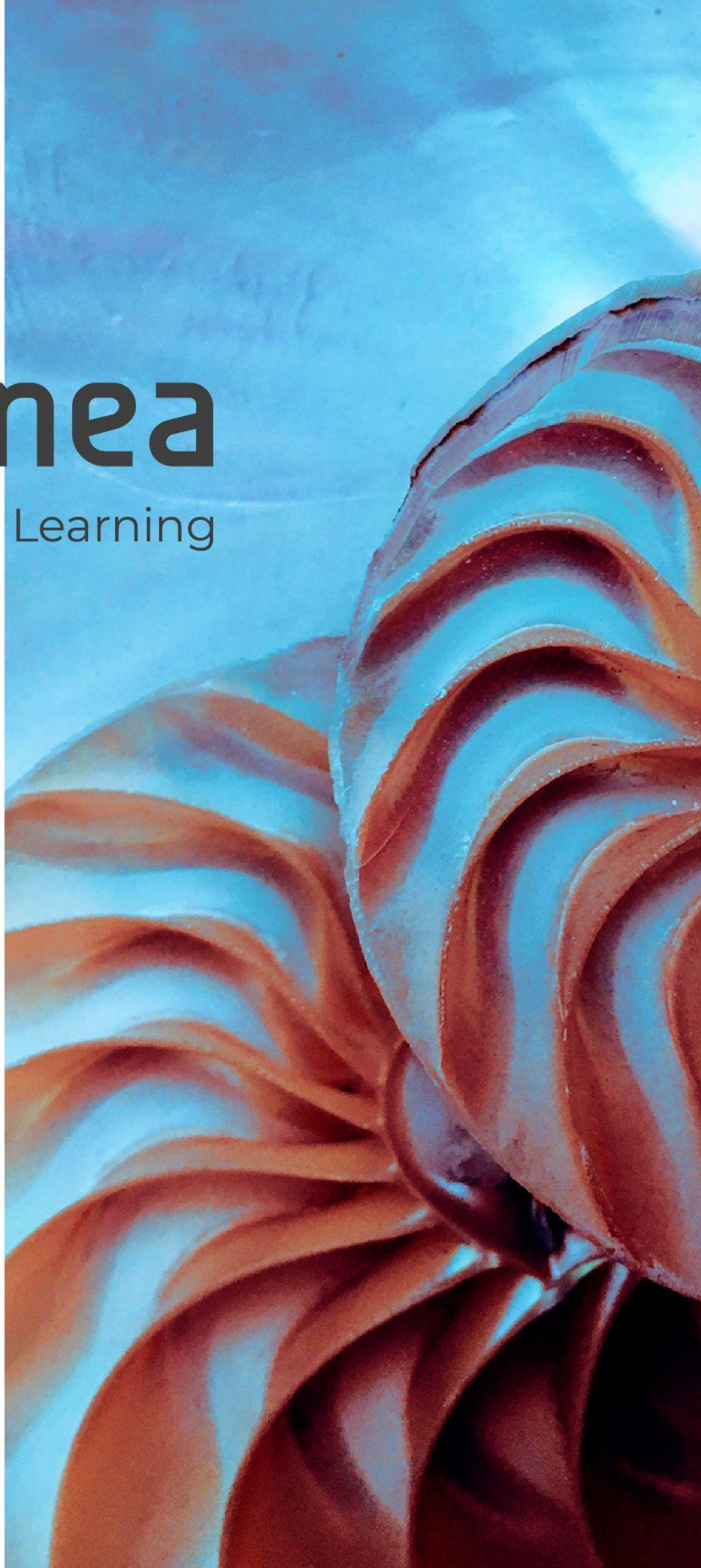
# Aitenea

Easy Machine Learning



www.aerin.es

# Documento de Instalación y uso de Altenea

# Índice.

# 1. Instalación de AItenea

## 1.1. Introducción

AItenea es un *framework* de *machine learning* de propósito general cuya principal virtud se encuentra en su capacidad para convertir cualquier algoritmo implementado en un elemento disponible en una capa de visualización para una programación visual mediante flujos. Con esta filosofía resulta mucho más directo pasar de una implementación teórica o de prueba de concepto a un bloque de funcionalidad preparado para la producción. Desde un punto de vista formal AItenea unifica las fases de preparación y transformación de datos, modelado, evaluación y despliegue, incorporando al mismo tiempo un mecanismo para añadir en esta cadena algoritmos nuevos desde su formulación matemática. De forma más descriptiva el objetivo de AItenea es poder disponer de un sistema capaz de poner en producción y/o ensayar modelos de *machine learning* sin necesidad de hacer codificación alguna, todo ello de forma muy intuitiva. Además el diseño de AItenea facilita la incorporación rápida de nuevos algoritmos por lo que no deberían existir límites evidentes al crecimiento y ampliación de métodos, por novedosos que estos resulten. Para alcanzar estos objetivos AItenea se compone de tres capas o niveles. A modo de introducción diremos que estos tres niveles son:

- Capa de bajo nivel o núcleo de la aplicación. Está conformada por una serie de clases y metaclases que permiten transformar los datos y realizar los modelos de m*achine learning*.

- Una capa intermedia, a modo de interfaz, entre la capa de bajo nivel y la capa superior.

- Una capa de alto nivel amigable a modo de interfaz gráfica de usuario para poder usar la funcionalidad de la capa de bajo nivel mediante programación de flujos.

## 1.2. Dependencias

Antes de desplegar el proyecto es necesario disponer de las siguientes dependencias:

- Disponer de la herramienta de control de versiones, Git (https://git-scm.com/doc).

  **sudo apt install git-all**

- Incluir el repositorio *ElasticTools* dentro de la carpeta raíz de AItenea a través del siguiente comando:

  **git clone https://github.com/AerinSistemas/ElasticTools.git**

- Instalar **make** en nuestro equipo con el siguiente comando:

  **sudo apt-get install build-essential**

- Instalar **bash**

  **sudo apt install bash-completion**

- Se requiere instalar Docker (si se quiere hacer un despliegue para desarrollo), y se recomienda seguir las instrucciones del desarrollador:

  https://docs.docker.com/engine/install/ubuntu/

  https://docs.docker.com/compose/install/

- Se debe comprobar si dentro del directorio raíz *aitenea* existe la carpeta data/csv. En caso de que la carpeta no existiese se deben seguir los siguientes pasos:
  - ➢ entrar dentro del directorio Docker/backend
  - ➢ abrir fichero Dockerfile e incluir las siguientes líneas:

    **RUN mkdir /opt/aitenea/data**

    **RUN mkdir /opt/aitenea/data/csv/**

## 1.3. Despliegue

Actualmente AItenea cuenta con tres opciones de despliegue:

- **Desarrollo**: Despliegue usando Docker y recarga de nodos automática al modificar su código.
- **Producción**: Despliegue sin Docker.
- **Producción en varios equipos**: Despliegue sin Docker, separando Node-RED y Django/Redis en 2 equipos diferentes.

El código desarrollado para este proyecto está disponible en el repositorio Git: https://github.com/AerinSistemas/Aitenea.

Para clonar el repositorio:

**git clone https://github.com/AerinSistemas/Aitenea.git**

### 1.3.1. Despliegue para desarrollo (con Docker)

Si deseamos desplegar la opción de desarrollo se deben seguir los siguientes pasos:

- Generar el entorno. Nos situamos dentro del directorio raíz de AItenea y ejecutamos:

  **make devel**

- Iniciar el entorno. Una vez ejecutado el paso anterior, usamos el siguiente comando:

  **make run**

Para un despliegue correcto, desde cero, previamente se deben borrar todos los contenedores, volúmenes e imágenes relacionados con AItenea.

### 1.3.2. Despliegue para producción (sin Docker)

Si deseamos desplegar la opción de producción se deben seguir los siguientes pasos:

- Generar entorno de producción. Situados dentro de la carpeta raíz de AItenea ejecutamos:

  **make production**

  Al ejecutar el *script* de despliegue, este nos preguntará por nuestro valores personalizados, si no queremos modificarlo, al pulsar *Enter* se asignan los valores por defecto (Figura 1.1 Despliegue de producción):

  - Django: 7000.

  - Redis: 6379.

  - Node-RED: 1880.

```
ramona@ramona-HP-Laptop-15s-fq1xxx:~/aitenea$ make production
Introduce el puerto externo disponible para (Django), por defecto (7000)
Introduce el puerto externo disponible para (Node-RED), por defecto (1880)
Introduce el puerto externo disponible para (Redis), por defecto (6379)
```

Figura 1.1 Despliegue de producción

- Iniciar Django y Redis. Situados en el directorio raíz ejecutamos:

  **make run-production-backend**

- Iniciar el entorno Node-RED. Situados en el directorio raíz AItenea ejecutamos:

  **make run-production-nodered**

### 1.3.3. Despliegue para producción en varios equipos

AItenea cuenta con una opción de despliegue en la cual la parte de Django/Redis y la parte de Node-Red se puedan desplegar en equipos diferentes Figura 1.2. Representación esquemática de despliegue en varios equipos.

El *script* pedirá introducir la IP pública del equipo donde estará desplegada la otra parte de AItenea:

- Al desplegar el entorno de Node-RED deberemos introducir la IP de Django/Redis.
- Al desplegar el entorno de Django/Redis debemos introducir la IP de Node-RED.

Cuando el *script* lo pida, deberemos definir los puertos externos que queramos usar, si no introducimos puertos se asignan los valores por defecto:

- Django: 7000.

- Redis: 6379.

- Node-RED: 1880.

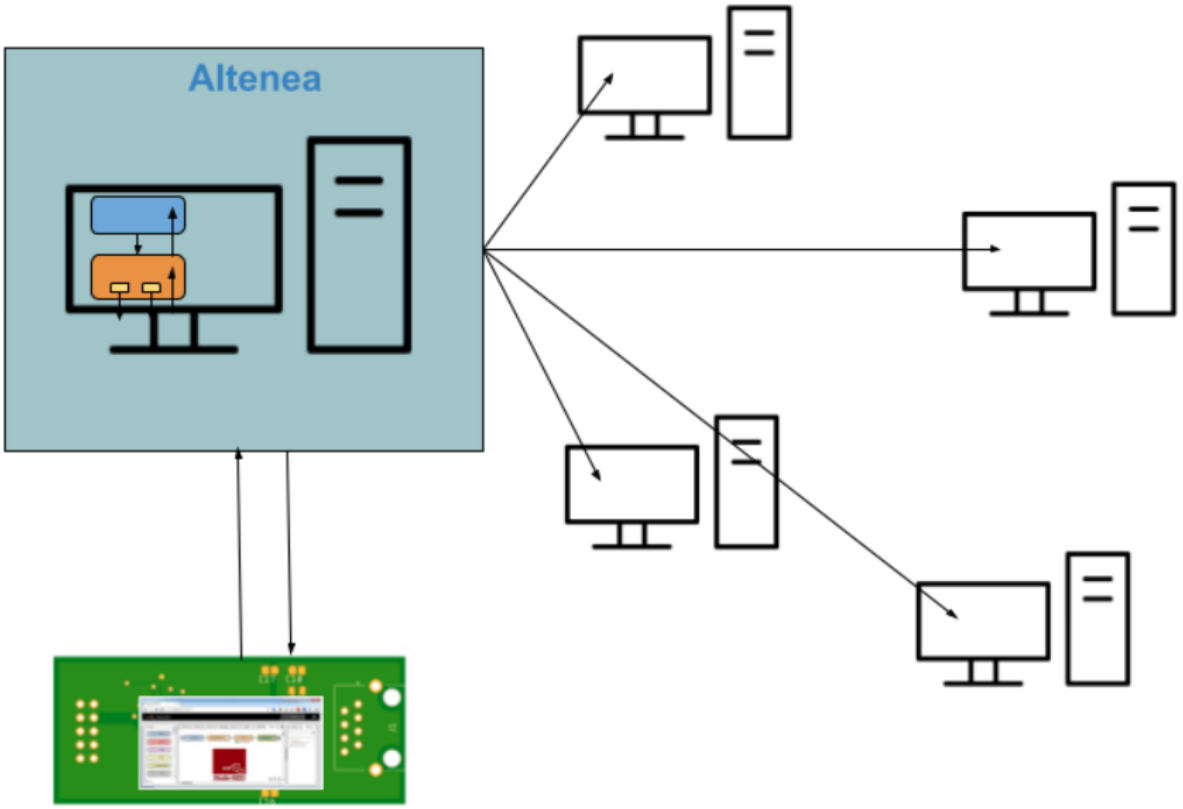



Figura 1.2. Representación esquemática de despliegue en varios equipos

Despliegue en el primer equipo:

- Generamos el entorno para Django/Redis con el siguiente comando:

  **make production-backend**

  Nos pedirá introducir IP y puertos del equipo.

- Iniciamos Django y Redis, a través del comando:

  **make run-production-backend**

Despliegue en el segundo equipo:

- Generamos el entorno para Node-RED con el siguiente comando:

  **make production-nodered**

  Nos pedirá introducir IP y puertos del equipo.

- Iniciamos Node-RED, a través del comando:

**make run-production-nodered**

## 1.4. Estructura del repositorio Altenea

Una vez que el usuario haya instalado AItenea siguiendo las indicaciones 1.3. Despliegue se encuentra una estructura de directorios como la que describimos a continuación:

*aitenea/*

|---- *aitenea_api*

|---- *aitenea_core*

|---- *aitenea_node-red*

|---- *docker*

|---- *docs*

|---- *elastictools*

|---- *exceptions*

|---- *logsconf*

|---- *scripts*

*__init__.py*

*.env*

*.gitingnore*

*.gitlab-ci.yml*

*babe.config.json*

*docker-compose.yml*

*makefile*

*package.json*

*Readme.md*

*redis.conf*

*webpack.config.js*

***aitenea_api***: representa la capa de negocio. Esta capa proporciona una API REST totalmente funcional que sirve como interfaz entre una aplicación de alto nivel y el núcleo o *aitenea_core*.

***aitenea_core***: incluye los elementos básicos que conforman la capa de bajo nivel y los métodos o algoritmos introducidos, ya sean de transformación o de *machine learning.*

***aitenea_node-red****:* Contiene la arquitectura de los nodos implementados específicamente para AItenea; los correspondiente ficheros .html, .js, package.json.

***elastictools***: Es la herramienta desarrollada a medida para conectar una base de datos Elasticsearch con AItenea. Su funcionalidad es gestionar los índices existentes en la base de datos.

***docs***: contiene los manuales explicativos de *aitene_core*, *aitene_api*, y *aitene_nodes*

***exceptions***: Incluye las clases auxiliares de AItenea para gestionar las excepciones.

***logsconf***: Incluye las herramientas del *logging* de AItenea.

***scripts***: Contiene todo los scripts de instalación de AItenea.

En los siguiente capítulos describiremos con un poco más de detalle la estructura de la capa de bajo nivel *aitenea_core*, y la capa de alto nivel *aitenea_node-red*.

# 2. Altenea en 30 minutos

Existen dos formas de usar Altenea: una Altenea para un uso amigable, que no requiere conocimientos de programación previos, Altenea Node-Desk; por otro lado un uso de Altenea como *framework*, cuyo fin es el de codificar nuevos algoritmos para que estos puedan usarse posteriormente en el Altenea Node-Desk. En este paseo por Altenea nos centraremos exclusivamente en Altenea Node-Desk, dejando su uso como *framework* para el capítulo "Guía de desarrollo".

## 2.1. Un paseo por Altenea Node-Desk

Una vez realizada la instalación, el acceso al Altenea-Desk se realiza a través del siguiente link: http://0.0.0.0:1880/, o en su defecto en la IP y puerto que el usuario ha escogido durante el despliegue de la herramienta.

Altenea Node-Desk está compuesto de varios elementos, como puede verse en Figura 2.1. Altenea Node-Desk:

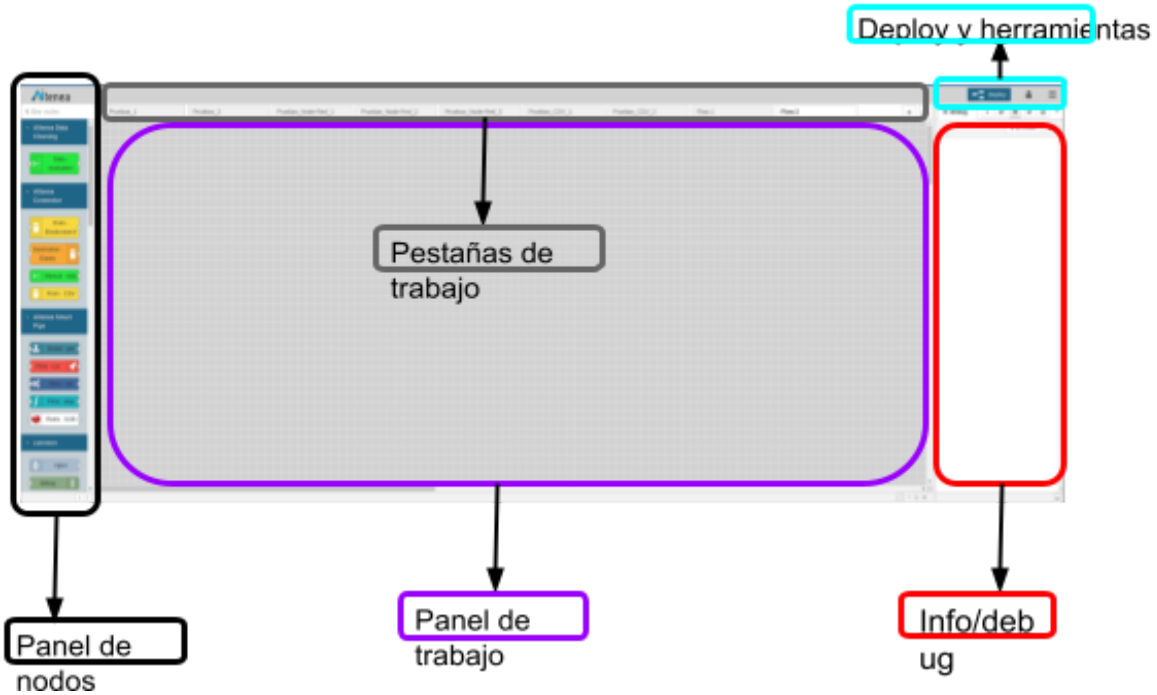



Figura 2.1. Altenea Node-Desk

- **El panel de nodos** (Figura 2.2. Panel de nodos): se encuentra en el lado izquierdo del lienzo y contiene nodos ya instalados y listos para usar. Aquí están visibles también los nodos específicos de AItenea. Los nodos están divididos en grupos, según su funcionalidad, hay varios grupos disponibles como por ejemplo: *common*, *function*, *network*, etc. Hay un repositorio disponible de donde se pueden instalar más nodos según necesidad. También se pueden construir nuevos nodos siguiendo la metodología de Node-Red.

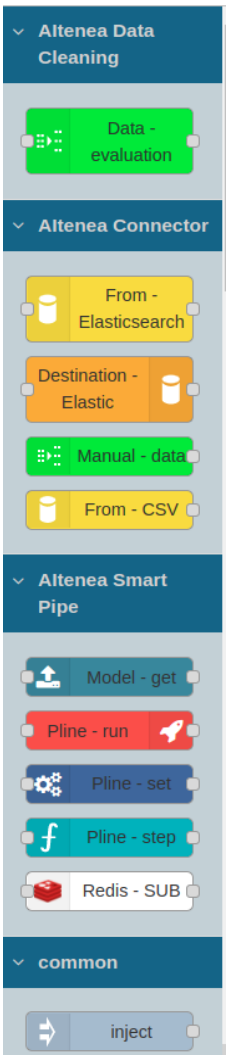



Figura 2.2. Panel de nodos

- **El panel de trabajo** (Figura 2.3. Panel de trabajo): es la parte central del lienzo donde se arrastran los nodos y se conectan para que realicen una tarea

concreta. Si dicha tarea puede ser reutilizada como subtarea en otro estudio, se pueden crear y guardar subflujos de la siguiente manera: seleccionando la opción *Subflow -> Create Subflow* en el menú, que creará un subflujo en blanco y lo abrirá en el espacio de trabajo y donde se irán arrastrando los nodos correspondientes, de esta manera se simplifica la visualización. Realizando un doble *click* sobre un nodo seleccionado, se accede a la información sobre dicho nodo y su configuración. En la parte superior existen las pestañas, para poder tener varios lienzos, para crear distintos flujos. En la parte inferior derecha existe una opción para dar o quitar zoom al lienzo. Es útil en el caso de tener flujos muy extensos.

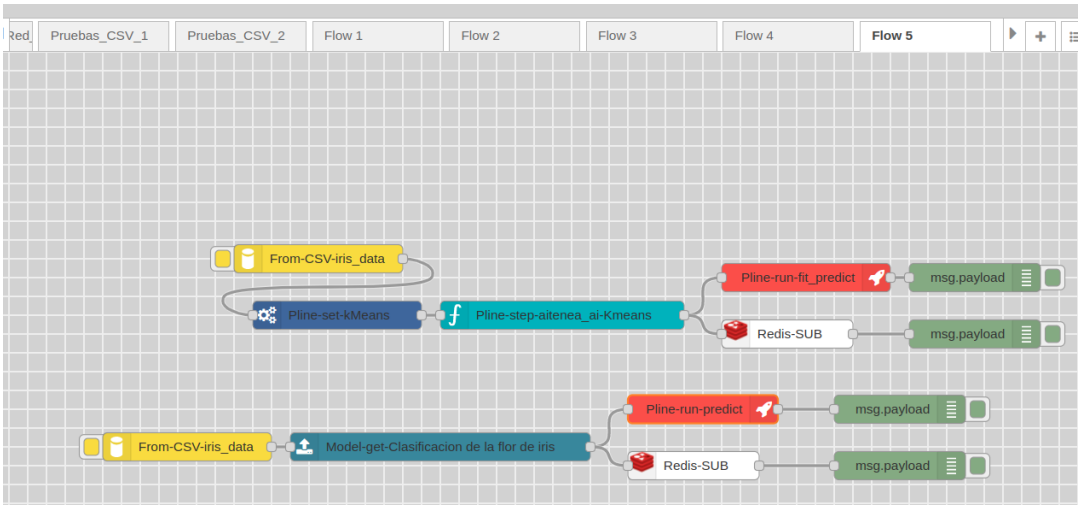



Figura 2.3. Panel de trabajo

- **El panel de información/*Debug*** (Figura 2.4. Panel de información): en la parte izquierda del lienzo, que ofrece la información sobre el nodo seleccionado. También ofrece información sobre la ejecución de nodos y otros datos de *Debug*.

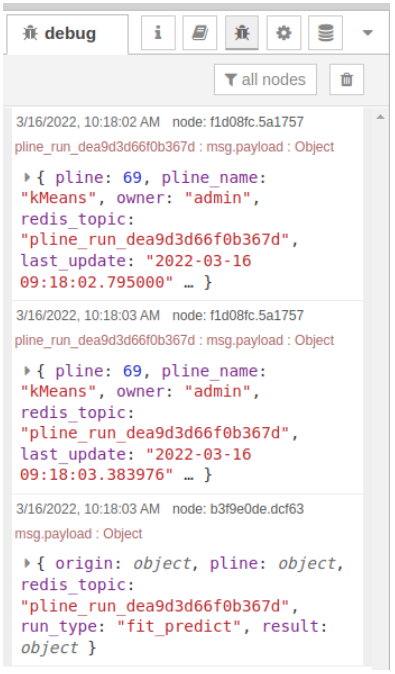



Figura 2.4. Panel de información

- **Pestañas de trabajo**: Los flujos pueden agruparse por pestañas, estas pestañas se pueden editar, borrar o crear al antojo del usuario. Las diferentes pestañas se muestran en la barra superior, justo encima del panel de trabajo.
- ***Deploy* y herramientas** (Figura 2.5. Botón Deploy): Justo encima del panel informativo se encuentra el *Deploy* y las herramientas de gestión. Las herramientas de gestión permiten crear agrupaciones de flujos, bloques que representan flujos y también añadir nodos propios de Node-Red desarrollados por la comunidad. El botón de *Deploy* es fundamental ya que debe ser pulsado para confirmar los cambios realizados en los flujos.

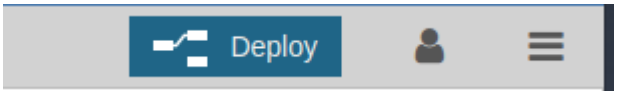



Figura 2.5. Botón Deploy

Este botón tiene dos tonos, azul para indicarnos que se han realizado cambios y que debe pulsarse para desplegarlos y gris oscuro, mostrando que el despliegue se ha realizado.

Altenea Node-Desk funciona según el paradigma de programación de flujo, es decir, las construcciones que realicemos se ejecutarán de izquierda a derecha, donde en el extremo izquierdo se encontrarán las entradas y a la derecha las posibles salidas. Altenea Node-Desk permite usar, en las entradas y en las salidas de la funcionalidad propias de Altenea

cualquier bloque de los proporcionados por Node-Red, siempre y cuando exista una coherencia. No obstante en los siguientes párrafos describiremos el uso exclusivo de los bloques de AItenea, vista la funcionalidad principal pondremos un ejemplo de coexistencia entre bloques de Node-Red y los de AItenea.

Comenzamos aquí una pequeña descripción de cómo construir una funcionalidad o tubería en AItenea Node-Desk.

## 2.2. Crear modelos

Supongamos que deseamos crear un clasificador (modelo clasificador), digamos por ejemplo para el conocido conjunto de datos flor Iris (o conjunto de datos iris de Fisher https://gist.github.com/netj/8836201). Lo primero que debemos tener es el propio conjunto de datos, posteriormente escoger el modelo de clasificación, y por último ejecutar el modelo.

Presentamos a continuación pasos a paso el proceso para construir nuestro clasificador.

### 2.2.1. Selección de los datos

En la versión actual de AItenea disponemos de tres formas de adquirir este conjunto: bien de un archivo CSV, donde la cabecera debe estar disponible en la primera fila; bien mediante un índice en la conocida base de datos Elasticsearch, pero para esta opción deberíamos tener en un servidor corriendo esta base de datos; y por último mediante un conjunto de datos introducidos a mano.

Describimos aquí únicamente las dos primeras posibilidades ya que la última solo es recomendable para un conjunto reducido de datos, para pruebas rápidas.

#### 2.2.1.1. Selección desde la base de datos

Para cargar un fichero desde una base de datos Elasticsearch se debe usar el nodo *From-Elasticsearch*, que describimos a continuación (Figura 2.6. Nodo From Elasticsearch).

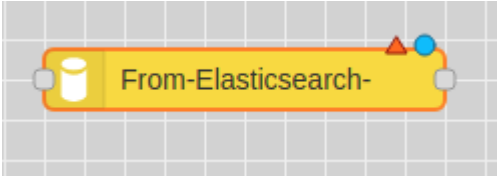


Figura 2.6. Nodo From Elasticsearch

Al desplegar el nodo se abre la siguiente ventana, donde se nos piden los datos de conexión a la base de datos (pulsando el símbolo lápiz se abre la ventana de conexión de la base de datos) y deberemos especificar el nombre del nodo en la caja *Node Name* Figura 2.7. Nodo From Elasticsearch, autentificación.

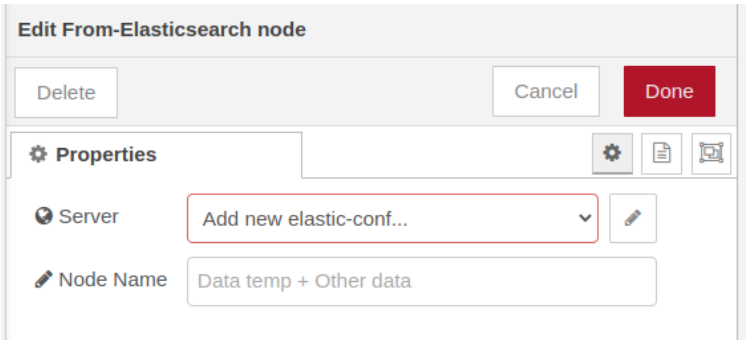



Figura 2.7. Nodo From Elasticsearch, autentificación

A continuación se selecciona el índice deseado del listado de índices disponibles en nuestra base de datos Figura 2.8. Nodo From Elasticsearch, selección de índices….

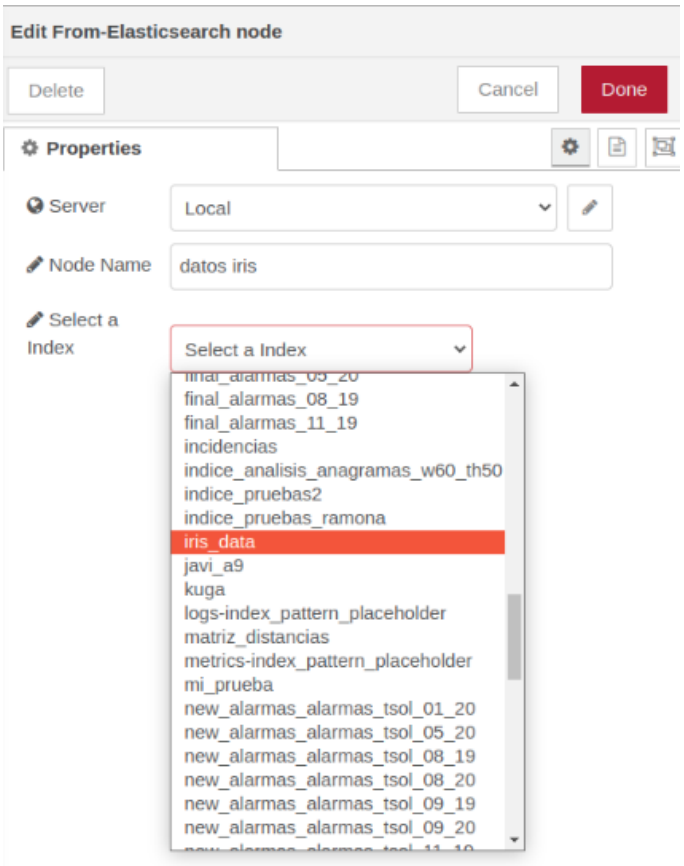



Figura 2.8. Nodo From Elasticsearch, selección de índices

Una vez seleccionado el índice, se despliega automáticamente el listado con las variables disponibles, entre las cuales elegimos las variables predictoras *X*, y la (las) variables objetivo *Y* Figura 2.9. Nodo From Elasticsearch, selección de variables.

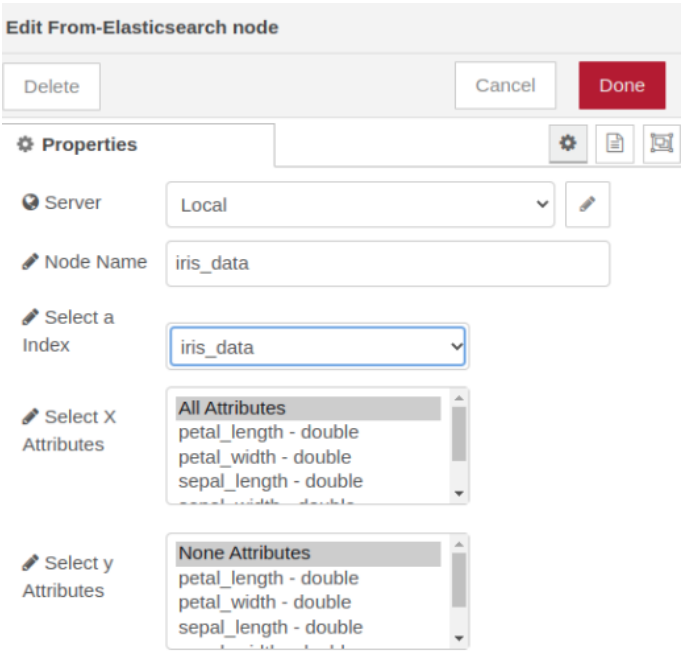



Figura 2.9. Nodo From Elasticsearch, selección de variables

Finalmente confirmamos la configuración pulsando *Done*, y nuestros datos ya están disponibles como datos de entrada de nuestro modelo.

### 2.2.1.2. Selección desde fichero CSV

En el ejemplo que nos ocupa nuestra elección para recuperar los datos es un archivo CSV por lo cual recurrimos al bloque *From*-CSV (Figura 2.10. Nodo From-CSV).

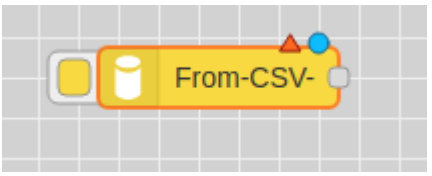



Figura 2.10. Nodo From-CSV

En la pantalla de despliegue se nos pide, como primer paso, nuestra autentificación ( introducir nuestros datos de usuario Figura 2.11. Nodo From-CSV, autentificación de…).

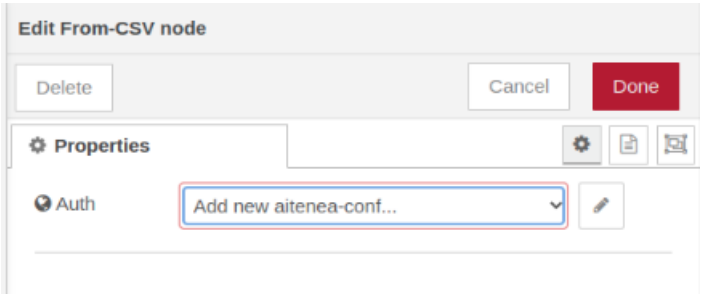



Figura 2.11. Nodo From-CSV, autentificación de usuario

Seguidamente se nos abre la siguiente ventana donde se nos pide: nombrar el índice en la opción *Index Name*), seleccionar nuestro fichero pulsando *Choose file*, y finalmente pulsando *Upload CSV* se nos muestra que el fichero se ha cargado correctamente Figura 2.12. Nodo From-CSV, selección y….

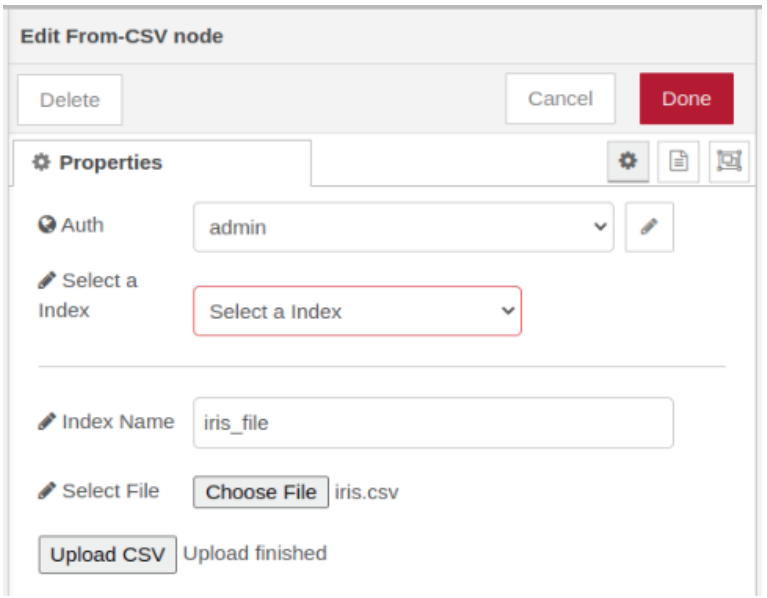



Figura 2.12. Nodo From-CSV, selección y carga del fichero

Seguidamente seleccionamos nuestro índice (*iris_file*) del listado disponible en la opción *Select a Index* Figura 2.13. Nodo From CSV, selección de índices.

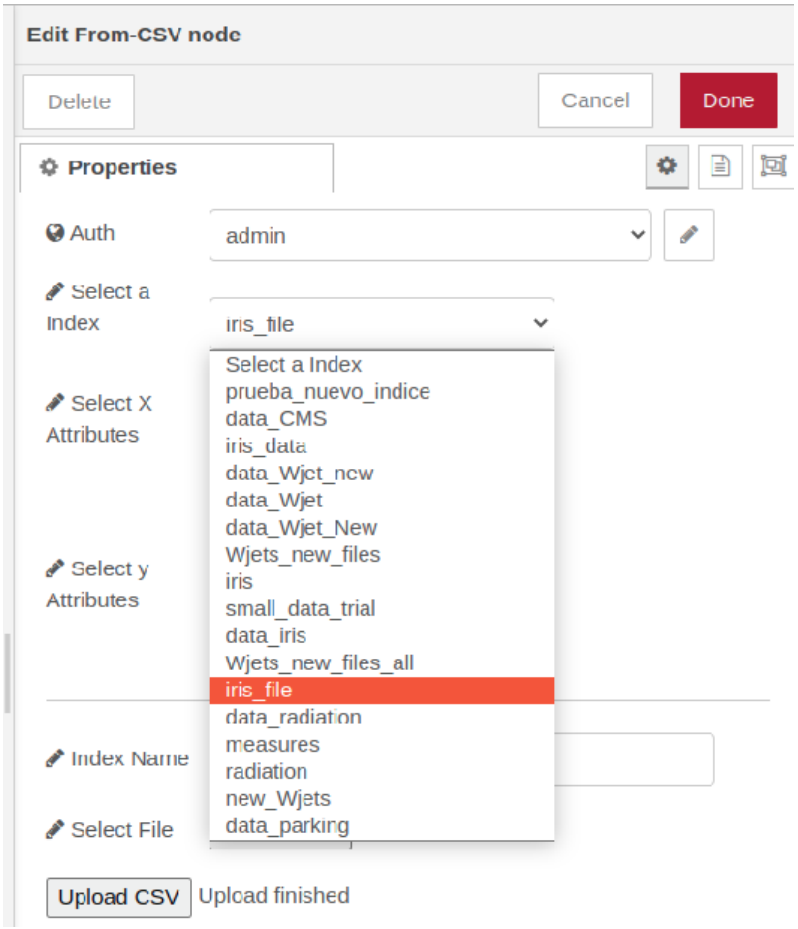



Figura 2.13. Nodo From CSV, selección de índices

Una vez seleccionado nuestro índice, se nos despliega automáticamente el listado de variables disponibles, entre las cuales elegimos las variables predictoras *X*, y la(las) variables objetivo *Y* Figura 2.14. Nodo From-CSV, selección de….

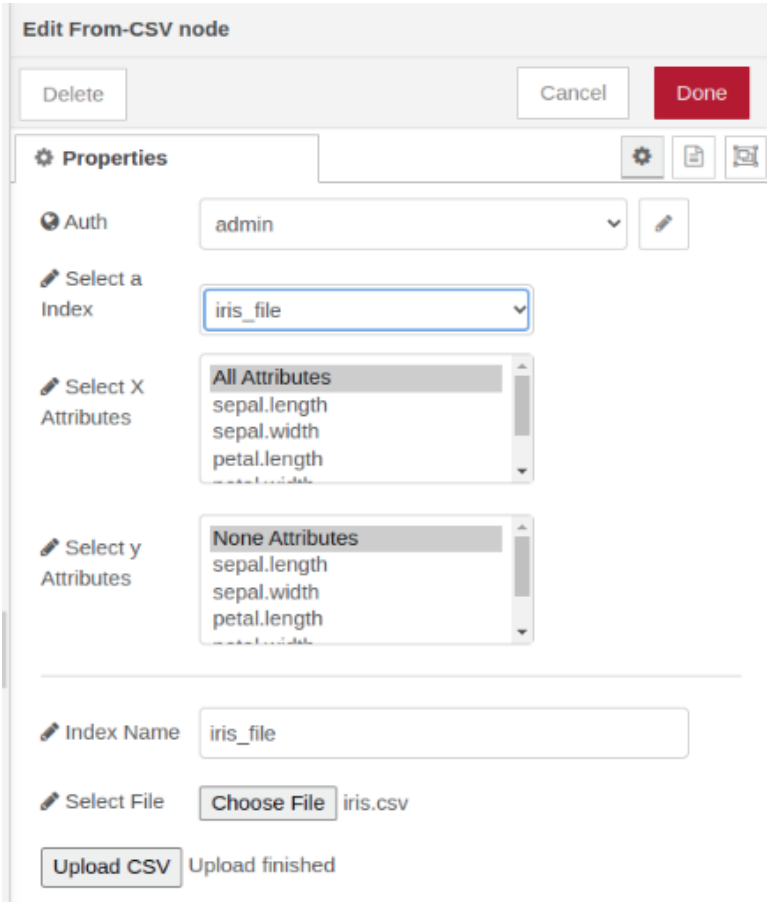



Figura 2.14. Nodo From-CSV, selección de variables

Finalmente confirmamos la configuración pulsando *Done*, y nuestros datos ya están disponibles como datos de entrada de nuestro modelo.

### 2.2.2. Configuración de una tubería

Una vez elegidos los datos de entrada así como los atributos de entrada (*X*) y el *target* (*y*), si lo tuviera, hay que configurar la tubería de funcionalidad. Una tubería es un flujo completo de funcionalidad que puede realizar cualquiera de las siguientes acciones:

1. Transformar datos según una o varias transformaciones realizadas en serie.
2. Transformar los datos y aplicar finalmente un modelo para entrenamiento.
3. Aplicar un modelo para entrenamiento sin transformar los datos.

Para construir una tubería es imprescindible configurarla ya que esta quedará registrada y guardada para reproducirla (re-entrenarla) en el momento que se deseé.


Figura 2.15. Nodo Pline-set

La tubería se configura a través del nodo *Pline-set* Figura 2.15. Nodo Pline-set. Como primer paso se nos pide nuestros datos de usuario Figura 2.16. Pline-set, autentificación de usuario…Seguidamente se nombra la tubería y se introduce una pequeña descripción de la misma. En el ejemplo que nos ocupa se trata de una clasificación de la flor de iris aplicando el algoritmo *Kmeans* Figura 2.17. Nodo Pline-set, opciones.

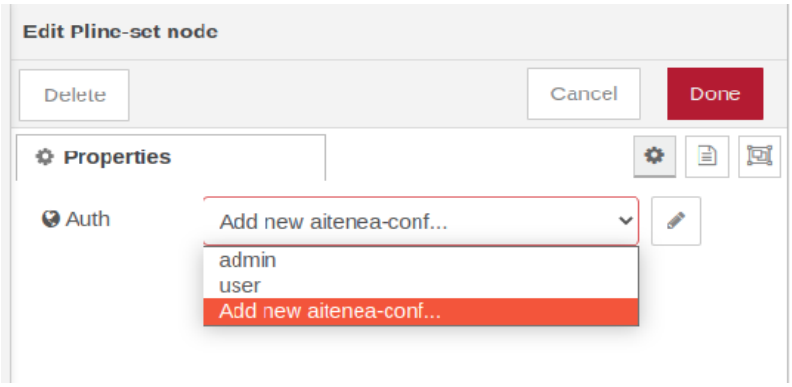



Figura 2.16. Pline-set, autentificación de usuario

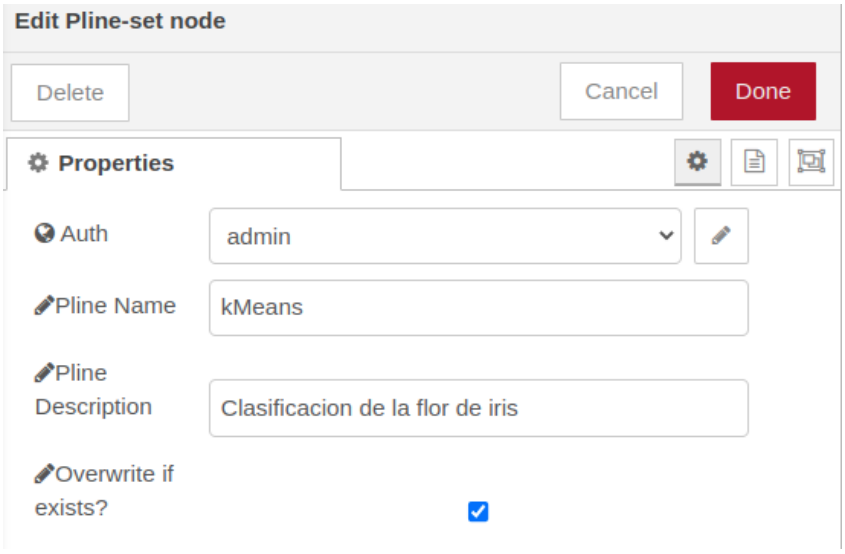



Figura 2.17. Nodo Pline-set, opciones

Obsérvese que existe la posibilidad de re-escribir la tubería, lo que evidentemente elimina la consecución de acciones de la anterior tubería.

#### 2.2.3.1. Selección de las tareas

Una vez configurada la tubería es necesario acumular acciones o pasos sobre los datos de entrada, una o varias transformaciones y/o un modelo final. Las acciones se realizan en serie, empezando por la izquierda y de forma consecutiva. Cada paso o acción se añade mediante el nodo *Pline-step* Figura 2.18. Nodo Pline-step.

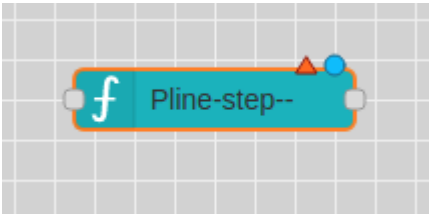



Figura 2.18. Nodo Pline-step

Al añadir uno de estos pasos se nos pedirá identificarnos con nuestros datos de usuario Figura 2.19. Nodo Pline-step, identificación de….

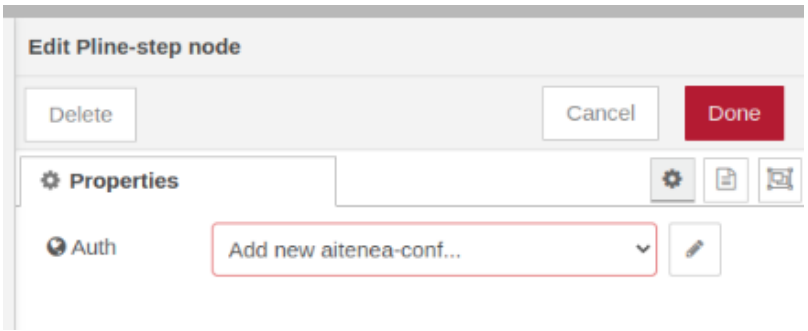



Figura 2.19. Nodo Pline-step, identificación de usuario

Seguidamente se nos abrirá una lista con todas las tareas implementadas en AItenea, que pueden ser de transformación (*aitenea_transform*) o de *machine learning* (*aitenea_ai*), en la opción *Select Class Type*, y se nos despliega la lista de algoritmos de tipo de la clase seleccionado Figura 2.20. Nodo Pline-set, selección de….

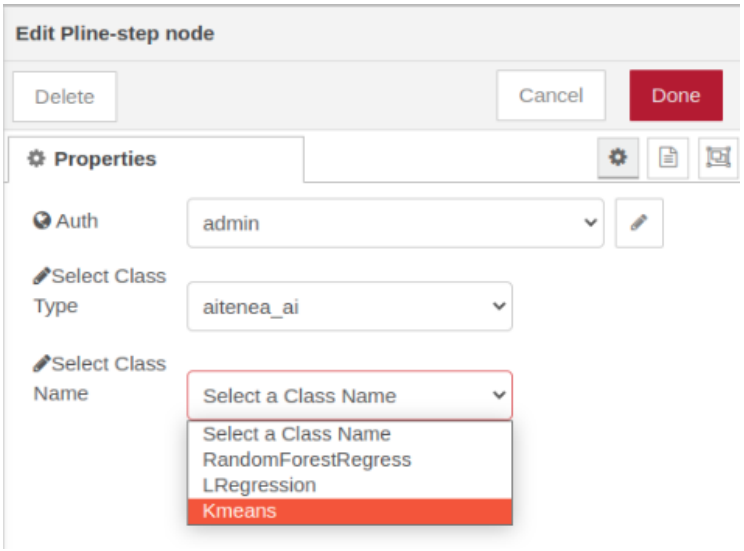



Figura 2.20. Nodo Pline-step, selección de clase, selección de algoritmo

Hay que tener en cuenta que **solo se permite un paso de *machine learning*** que **debe colocarse como paso final**. Por el contrario, las transformaciones son ilimitadas.

Para este ejemplo no añadimos transformación alguna, únicamente un algoritmo de *machine learning*, por ejemplo un *Kmeans*, en la opción *Select Class Name*, y automáticamente se cargan los parámetros de dicho algoritmo, en nuestro caso: *num_cluster*, *method*, etc Figura 2.21. Nodo Pline-step, carga de…Estos parámetros tienen un valor por defecto o un rango con posibles valores aceptados por el mismo, para que el usuario escoja los más adecuados para su caso de estudio.

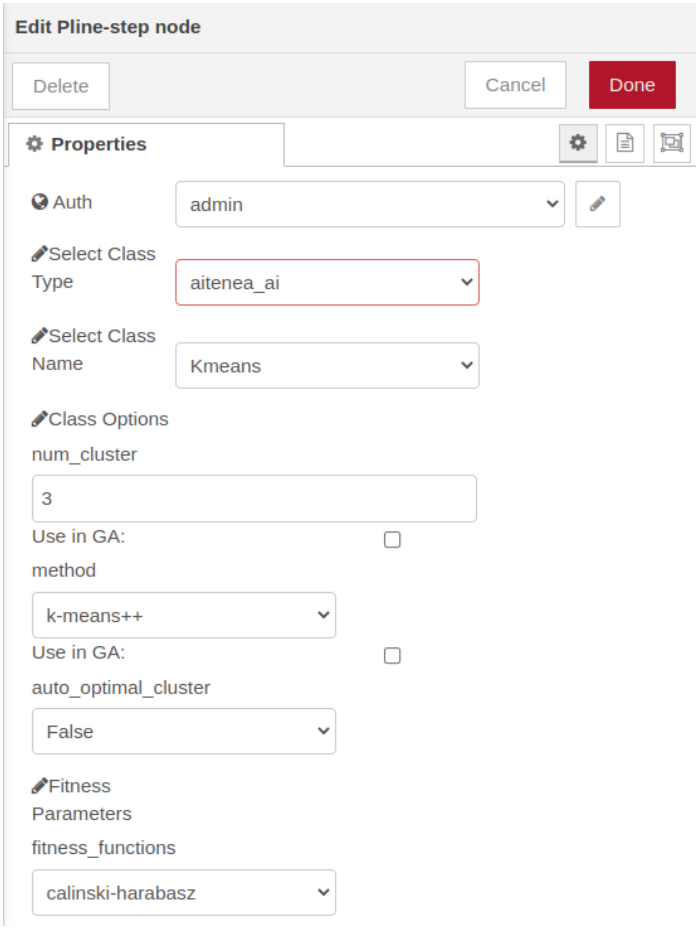



Figura 2.21. Nodo Pline-step, carga de parámetros del algoritmo

Como ya hemos mencionado la tubería puede contener tantas tareas de transformación como se considere necesario pero solamente una tarea de *machine learning* y debe ser la última.

### 2.2.4. Ejecutar la tubería

Una vez que se haya configurado la tubería seleccionando las tareas o pasos a aplicar sobre los datos de entrada Figura 2.22. Nodo Pline-run, se debe escoger la acción que se debe realizar sobre la misma (*fit*, *fit_transform*, *fit_predict*, *predict*), a través del nodo *Pline-run* Figura 2.23. Nodo Pline-run,selección de….

Figura 2.22. Nodo Pline-run

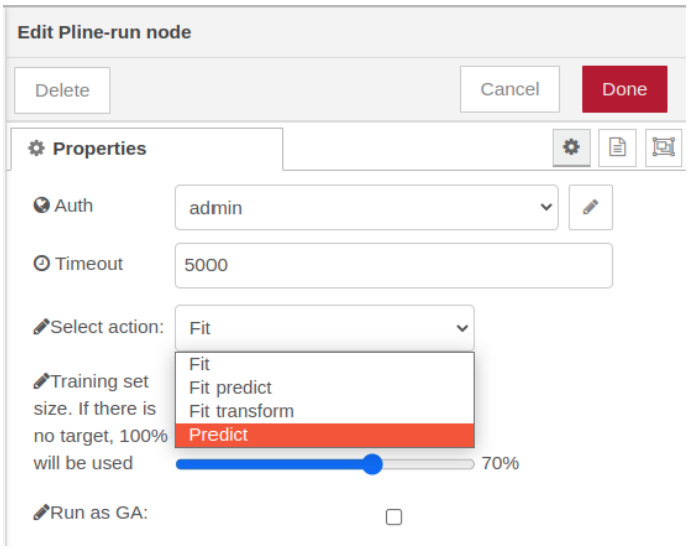



Figura 2.23. Nodo Pline-run, selección de la acción a realizar

Todos los modelos de *machine learning* aprenden de los datos con los que los entrenamos y a partir de ellos, intentan encontrar o inferir el patrón que les permita predecir el resultado para un nuevo caso. Para poder calibrar el modelo necesitaremos probarlo con un conjunto de datos que el modelo no haya visto durante el entrenamiento. Es por ello que en todo proceso de aprendizaje automático, los datos de trabajo se dividen en dos partes: datos de entrenamiento y datos de prueba o test. La opción *Training set size* en nuestro nodo nos permite dividir el set de datos de entrada Figura 2.24. Nodo Pline-run, selección de….

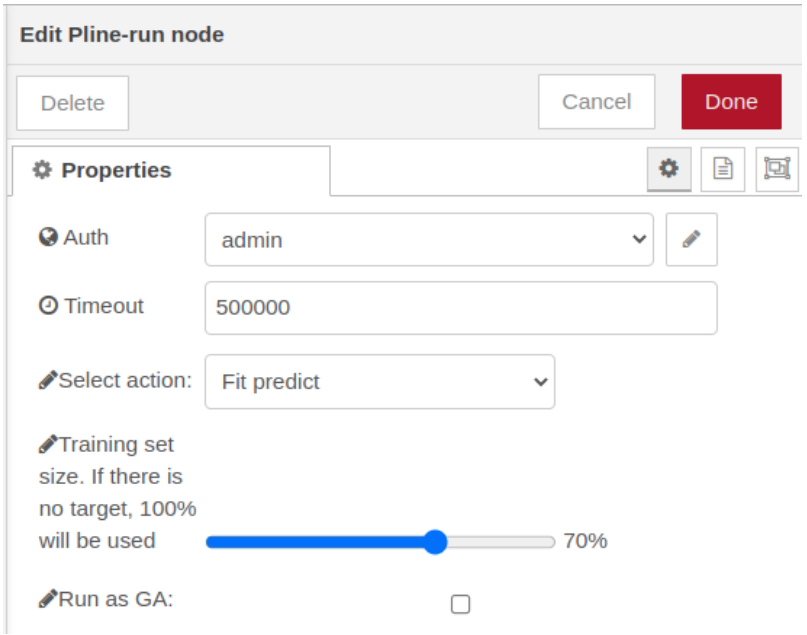



Figura 2.24. Nodo Pline-run, selección de test and training data

### 2.2.5. Informe de ejecución

El nodo *Redis-SUB* Figura 2.25. Nodo Redis-SUB ofrece un informe sobre el estado de ejecución de la tubería, gestionados por la API Reports 3.2.2. Reports, que se muestra en el panel lateral derecho, en la pestaña *Debug*.

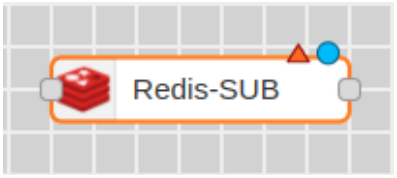



Figura 2.25. Nodo Redis-SUB

Desplegando cada uno de los mensajes se nos presenta información útil sobre la tubería, el resultado de la misma, y se nos informa si ha ocurrido algún error durante la ejecución.


Figura 2.26. Pestaña de información

La información ofrecida Figura 2.26. Pestaña de información por el nodo consiste básicamente en un *beatheart* de AItenea con el tiempo de ejecución, además de las características de la tubería en ejecución. Cuando se está empleando un método basado en algoritmos genéticos el informe podrá incluir el resultado de la función de *fitness* de cada individuo.

No es estrictamente necesario intercalar un nodo de reporte, pero ayuda a saber qué está pasando dentro de AItenea cuando se ejecuta una tubería. Hay que tener en cuenta que algunos métodos pueden demorar mucho, especialmente en estos casos es muy útil el uso de estos nodos informativos.

El nodo de informe de ejecución debe colocarse en paralelo junto al bloque de ejecución, tal y como se sugiere en la siguiente imagen Figura 2.27. Conexión del nodo Redis-SUB:

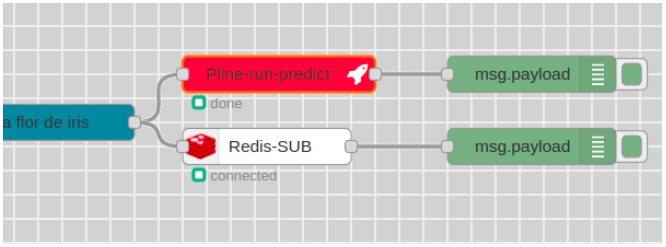



Figura 2.27. Conexión del nodo Redis-SUB

Igualmente es necesario colocar un bloque de *Debug* tras el nodo de informe si queremos visualizar las salidas que este nos ofrece.

## 2.3. Cargar modelos

Una vez que hemos entrenado una tubería (según hemos visto en el apartado 2.2. Crear modelos), incluso aunque esta solo consista en una serie de transformaciones, podemos recuperarla para ejecutarla de forma recurrente con nuevos datos. Por ejemplo, en el caso de clasificación de la flor de iris, el modelo alimentado con nuevos datos nos permitirá predecir resultados. Cada una de las acciones de transformación de los datos que tenga la tubería se van a realizar secuencialmente por lo que no va a ser necesario transformar previamente los mismos. Como puede verse es un método rápido de desplegar modelos de *machine learning*.

AItenea dispone de un nodo que permite cargar un modelo ya existente en AItenea, este bloque se denomina *Model-get* Figura 2.28. Nodo Model-get.

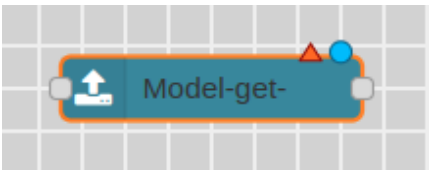



Figura 2.28. Nodo Model-get

Volvamos a nuestro ejemplo de clasificación de la flor de iris, y supongamos que necesitamos 'recuperar' el modelo y realizar una nueva predicción, porque disponemos de datos actualizados para nuestro modelo de estudio Figura 2.29. Ejemplo de tubería.

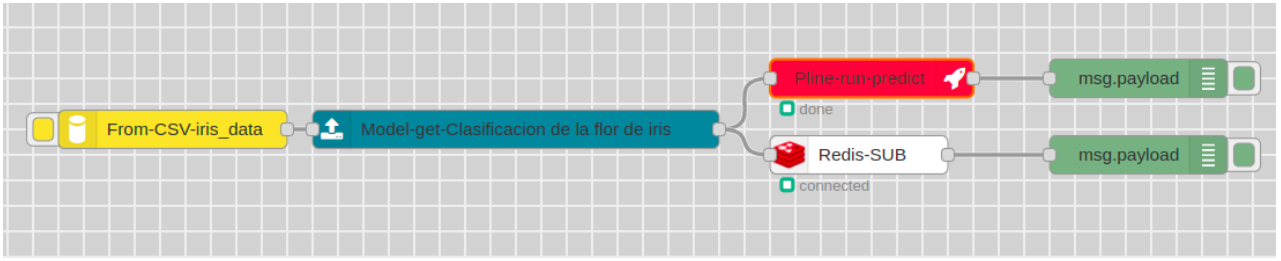



Figura 2.29. Ejemplo de tubería

A continuación describimos cada uno de los bloques que constituyen una tubería de estas características.

### 2.3.1. Fuente de datos

Cargamos nuestro fichero de datos a través de alguna de las fuentes de datos disponibles, tal como se ha detallado en 2.2.1. Selección de los datos. Estos datos sólo deberán cumplir un requisito, contener los mismos atributos que se usaron para entrenar el modelo.

### 2.3.2. Carga de modelos

Todas las tuberías entrenadas se encuentran guardadas en AItenea y son accesibles mediante el nodo *Model-get*. En la primera ventana de despliegue se nos pide nuestra autentificación Figura 2.30. Nodo Model-get, identificación de….

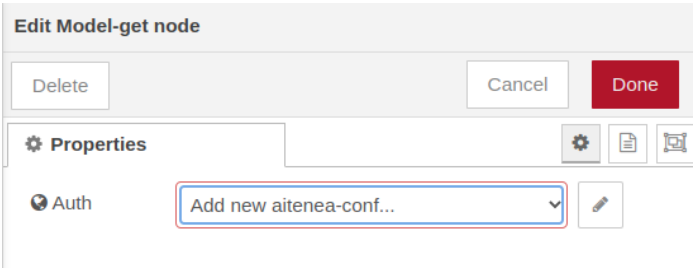



Figura 2.30. Nodo Model-get, identificación de usuario

Posteriormente se nos abre la ventana de selección del modelo (*Select pLine Name*), entre los ya existentes en AItenea Figura 2.31. Nodo Model-get, selección del….

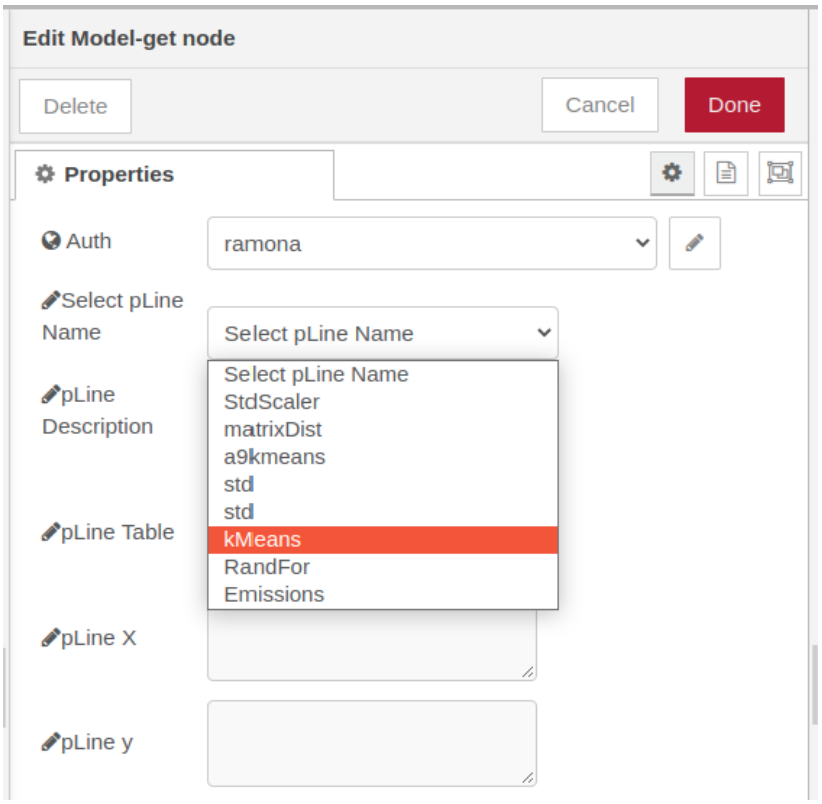



Figura 2.31. Nodo Model-get, selección del modelo a cargar

En nuestro caso seleccionando el modelo *kmeans* se carga automáticamente la información del mismo: la descripción, el nombre del fichero de datos de entrada, las variables predictoras (*pLine X*) y objetivo (*pLine y*) utilizadas para crear el modelo Figura 2.32. Nodo Model-get, despliegue de….

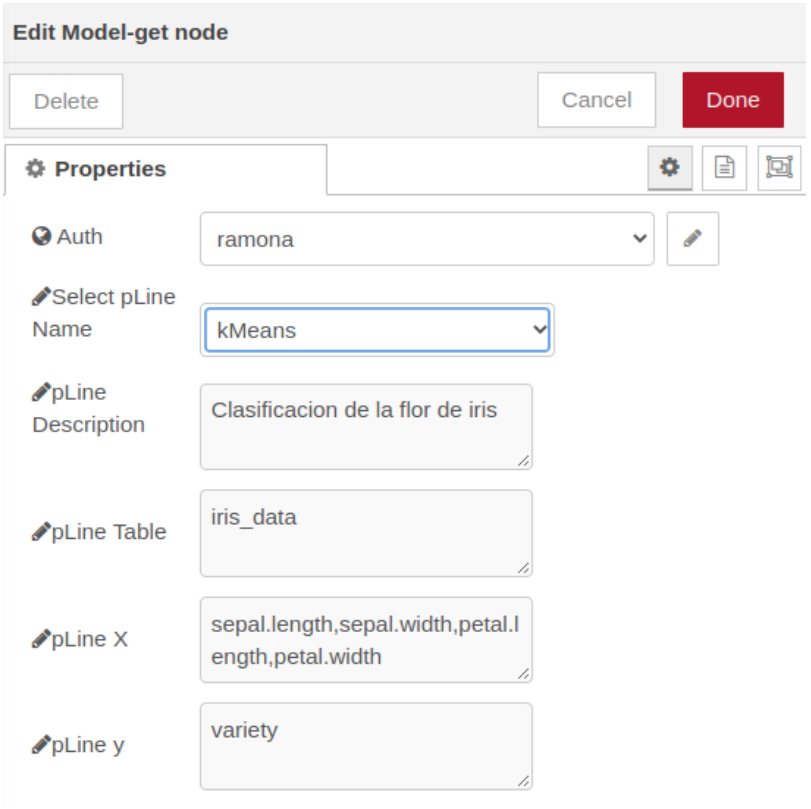



Figura 2.32. Nodo Model-get, despliegue de la información del modelo

### 2.3.3. Ejecutar el modelo

Se ejecuta el modelo a través del nodo Pline-run, realizando en este caso un *predict* para hacer una nueva predicción con el set de datos actualizado. Se nos pide primeramente autentificarse para posteriormente seleccionar la acción deseada Figura 2.33. Nodo Pline-run, selección de….

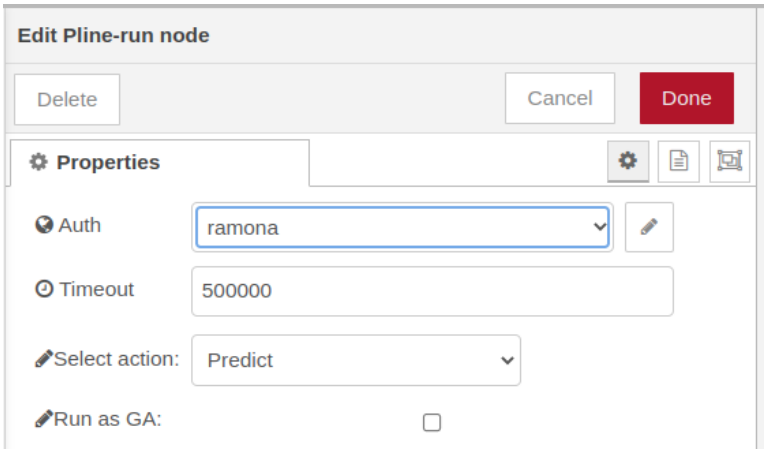


Figura 2.33. Nodo Pline-run, selección de la acción a realizar

# 3. Fundamentos

En este capítulo describimos con más detalle la capa de bajo nivel y la capa de alto nivel de AItenea. También se hace una descripción detallada del panel de control de AItenea, explicando sus funcionalidades.

## 3.1. Capa de bajo nivel

La capa de bajo nivel, denominada *aitenea_core*, está formada por las diferentes funcionalidades, clases y metaclases que conforman la funcionalidad central del sistema. En ella encontramos unas clases base que deberán ser heredadas por cualquier algoritmo que se desee incorporar, también encontraremos los elementos que permiten incorporar las tuberías para ejecución de los algoritmos.

### 3.1.1. Estructura de la capa de bajo nivel

La capa de bajo nivel ha sido organizada con la siguiente estructura de directorios:

*|---- aitenea_core*

*|---- aitenea_transform*

*|---- clustering*

*|---- ensemble*

*|---- linear_models*

*base_class_ai.py*

*base_class_preprocessing.py*

*decorators.py*

*perpetuity.py*

*pfactory.py*

*Aitenea_core* incluye los elementos básicos que conforman la capa de bajo nivel y los métodos o algoritmos introducidos, ya sean de transformación o métodos de *machine learning.* Estos algoritmos requieren ser introducidos dentro de su propio directorio y deben estar conformados por una clase principal que herede de *base_class_preprocessing.py* o de *base_class_ai.py*, este directorio puede contener otros archivos auxiliares (clases o conjuntos de métodos) si se considera necesario y siempre para uso interno de la clase principal. Es obligado escribir estos archivos auxiliares precedidos de un guión, al igual que los métodos que contenga, el propósito siempre es distinguir claramente entre los métodos perceptivos y los auxiliares. Si ya existe un directorio con algoritmos cuya tipología fuese similar al que se desea implementar, se puede añadir en el mismo directorio el fichero con la clase correspondiente.

Dentro del directorio *aitenea_transform* están agrupados todos los algoritmos de transformación implementados en AItenea.

El directorio *clustering* agrupa los algoritmos de *machine learning* de clusterización, p. ej. *Kmeans*.

El directorio *ensemble* agrupa los algoritmos de *machine learning* llamados algoritmos de conjunto, p. ej. *Random Forest*.

El directorio *linear_models* agrupa los algoritmos de *machine learning* que describen la relación lineal entre las variables del modelo, p. ej. regresión lineal.

*Base_class_preprocessing.py* es la clase base que se encarga de establecer las reglas imprescindibles para realizar una clase cuya funcionalidad es la de transformar datos. Se trata de una clase madre de la que deben heredar todas las clases que se implementen para transformar datos, las implementadas en *aitenea_transform*.

*Base_class_ai.py* es la clase base de la que heredarán todas las clases que implementan algoritmos de *machine learning*, con la finalidad de que las clases de inteligencia artificial deban construirse de una forma determinada usando métodos abstractos.

*Pfactory.py* es la clase base que tiene la función de construir tuberías. Las tuberías son elementos capaces de aplicar de forma secuencial una lista de transformaciones unidas a un modelo o estimador al final de la misma. Es necesario que los pasos intermedios no contengan modelos, únicamente transformaciones. En algún caso puede resultar interesante solo unir transformaciones sin estimador final, esta última opción también debe ser contemplada.

*Perpetuity.py* tiene la funcionalidad de gestionar los modelos de AItenea: listar los modelos creados, guardar modelos, o cargar un modelo previamente creado.

*Decorators.py* es una función que asegura la coherencia, entre la entrada y la salida de los datos en cada tarea (paso) de la tubería.

Todos los algoritmos implementados deben superar una serie de pruebas unitarias y pruebas de integración. A tal efecto en el directorio de cada algoritmo se encuentra un directorio *test*

donde se encuentran toda las pruebas que se han realizado, de los cuales se presentan a continuación para cada algoritmos, algún ejemplo.

Todos los algoritmos implementados en esta capa a nivel de código, tienen su representación gráfica en el nodo Pline-step sin que el programador tenga que hacer ningún esfuerzo de codificación. En Figura 3.1. Listado, en el nodo Pline-… se muestran algunos algoritmos implementados en AItenea.

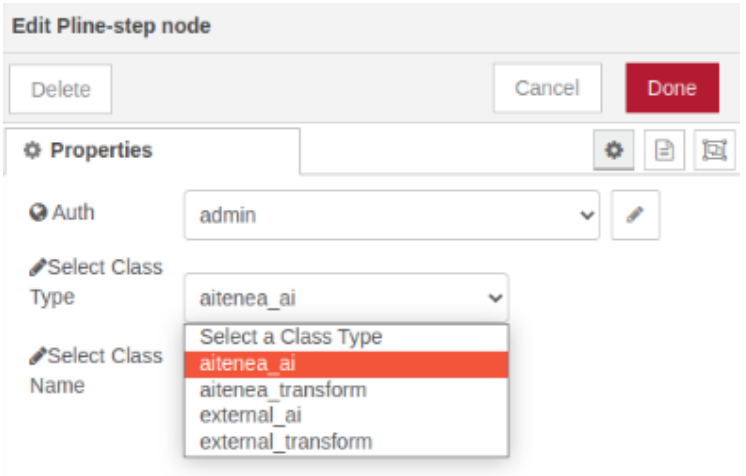



Figura 3.1. Listado, en el nodo Pline-step, de los tipos de algoritmos implementados en AItenea

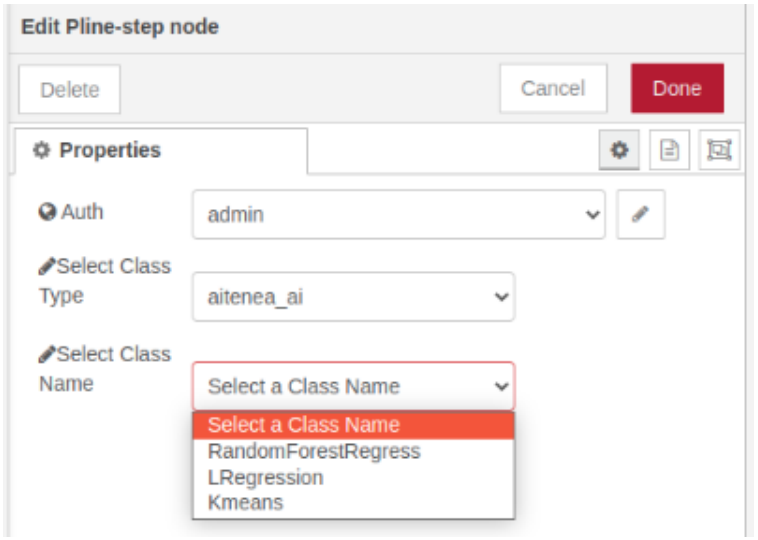



Figura 3.2. Listado, en el nodo Pline-step, de los algoritmos implementados en AItenea

### 3.1.2. Algoritmos de transformación

#### 3.1.2.1. StandardScaler

Es un algoritmo que realiza una transformación de los datos que se le pasan como entrada. Cuando el conjunto de datos de estudio contiene variables que son diferentes en escala, es recomendable transformar los mismos eliminando la media y escalando los datos de forma que su varianza sea igual a 1. La transformación se realiza de forma independiente en cada variable mediante el cálculo de las estadísticas relevantes en las muestras en el conjunto de entrenamiento. La media y la desviación estándar se almacenan para ser utilizadas posteriormente a través del método *transform*.

Mostramos a modo de ejemplo cómo usar este algoritmo importando el módulo correspondiente y pasando los parámetros de la clase de forma adecuada, como un diccionario. Usamos un set de datos artificial cuya transformación es conocida de antemano. A través de la función test de pandas *asert_frame_equal* comprobamos que el resultado que devuelve el algoritmo corresponde con el esperado.

Este algoritmo puede ser usado, si así se considera, como paso previo a un modelo de *machine learning*.

```
import dask
import pandas as pd
from aitenea.aitenea_core.aitenea_transform.scaler import StdScaler
from aitenea.aitenea_core.pfactory import PFactory
from sklearn.datasets import make_blobs
from pandas._testing import assert_frame_equal
from aitenea.exceptions import exceptions


def test_classScaler():
   """we test that class is created without error, it transforms data and compare with expected
result
   """
   datax = [[0, 0], [0, 0], [1, 1], [1, 1]]
   X = dask.array.from_array(datax)
   X = dask.dataframe.from_array(X)
   X = X.repartition(partition_size='100MB')
   expected_result = [[-1.0,-1.0],[-1.0,-1.0],[1.0,1.0],[1.0,1.0]]
   expected_result = pd.DataFrame(expected_result)
   class_group = "aitenea.aitenea_core.aitenea_transform.scaler"
   options = {"with_mean": True, "with_std": True}
   parameters = {"options": options}
   std = StdScaler(parameters)
   assert_frame_equal(std.fit_transform(X).compute(),expected_result)
```

### 3.1.3. Algoritmos de machine learning

#### 3.1.3.1. Kmeans

Es un algoritmo de clasificación no supervisada (clusterización) que agrupa las variables en *k*-grupos basándose en la similitud de sus características. La agrupación se realiza minimizando la suma de distancias entre cada variable y el centroide de su grupo o cluster. Se suele usar la distancia cuadrática. El algoritmo tal como está implementado en AItenea, incluye una mejora, contiene un método para calcular el número óptimo de clases k.

Mostramos una representación esquemática en la Figura 3.3. Representación esquemática del algoritmo Kmeans.

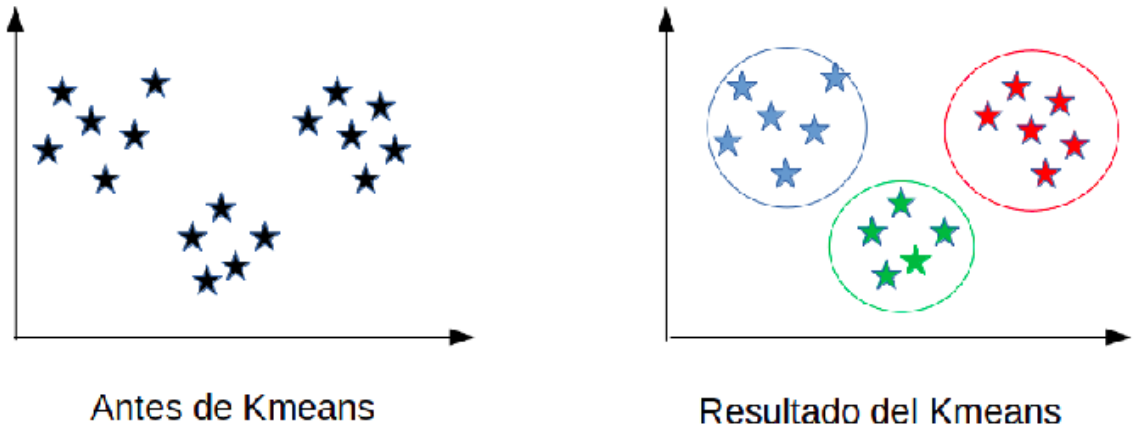



Figura 3.3. Representación esquemática del algoritmo Kmeans

En el ejemplo que proponemos a continuación verificamos que el algoritmo de clasificación asigna correctamente los datos de entrada, generados artificialmente, a una de las dos clases definidas, 0 o 1.

```
from aitenea.aitenea_core.clustering.kmeans import Kmeans
import dask
from numpy.testing import assert_array_equal

meth1 = "k-means||"
meth2 = "k-means++"

def test_clasification():
    """The test try kmeans clasification.
       The test is passed if the class is created without error and make a correct prediction
    """
    X = dask.array.from_array([[0, 0], [0.5, 0], [0.5, 1], [1, 1]])
    options = {"num_cluster": 2, "method": meth1, "auto_optimal_cluster": False}
    parameters = {"options": options}
    test = Kmeans(parameters)
    assert_array_equal(test.fit_predict(X).compute(), [0, 0, 1, 1])
```

### 3.1.3.2. Random Forest

El método de *Random Forest* (Bosque aleatorio) pertenece a la clase de métodos de conjunto, que combina las predicciones de varios estimadores base construidos con un algoritmo de aprendizaje dado para mejorar la robustez y la generalización del algoritmo.

El bosque aleatorio es un estimador que se ajusta a una serie de árboles de decisión de regresión en varios subconjuntos del conjunto de datos y utiliza el promedio para mejorar la precisión predictiva y controlar el sobreajuste. La principal ventaja es la mejora del rendimiento de generalización durante el entrenamiento, que se consigue compensando los errores de las predicciones de los distintos árboles de decisión.

En la Figura 3.4. Representación esquemática del algoritmo Random Forest, mostramos una representación esquemática del algoritmo. Cada árbol individual en el bosque aleatorio realiza una predicción de clase y la clase con más votos se convierte en la predicción de nuestro modelo.

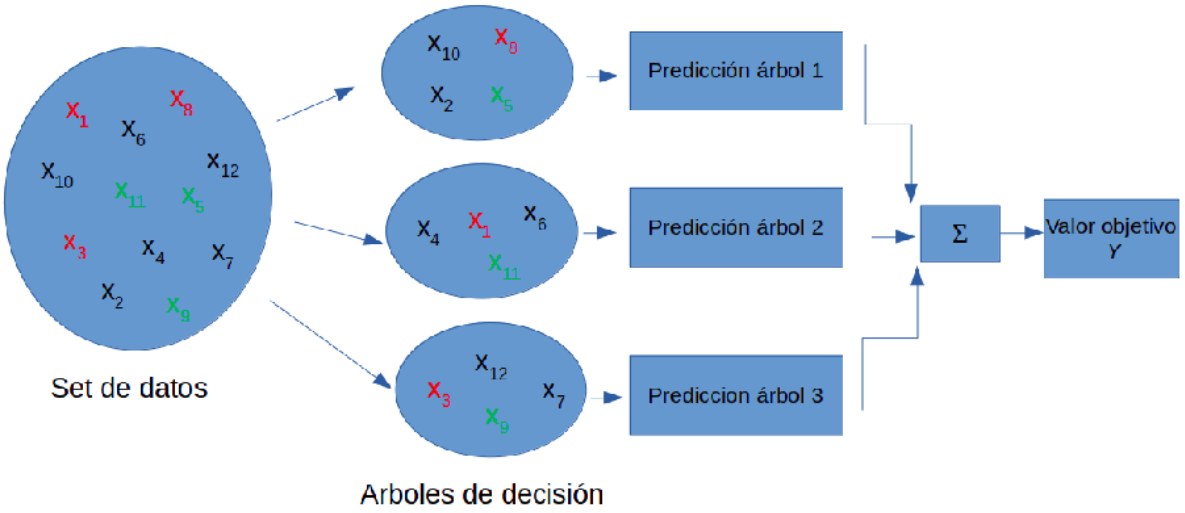



Figura 3.4. Representación esquemática del algoritmo Random Forest

En el ejemplo que proponemos, y usando el algoritmo *Random Forest*, verificamos otro elemento importante en AItenea que es la correcta gestión de las excepciones, que son errores específicos que es necesario controlar. El test que proponemos lanza el mensaje de *NotFitError* si se pretende hacer una predicción con el modelo, sin haberlo previamente entrenado con *fit*. Solamente a partir de los parámetros generados en el método *fit*, se puede realizar una predicción sobre nuevos datos presentados al algoritmo y se devuelve el resultado de la predicción.

```
from aitenea.aitenea_core.ensemble.random_forest import RandomForestRegress
from numpy.testing import assert_array_equal
from dask_glm.datasets import make_regression
from aitenea.exceptions.exceptions  import  NotFitError


def test_predictRF():
    """In this function we test that NotFitError is launched if the model is not fitted before
making prediction.
    """
    X, y = make_regression(n_samples=10, n_features=4, n_informative=2, chunksize=1)
    options = {"n_estimators":1000, "max_depth": None,
'max_features':'auto','min_samples_leaf':1,'min_samples_split':2}
    parameters = {"options": options}
    forest = RandomForestRegress(parameters)
    try:
        forest.predict(X)
    except NotFitError as err:
        assert True
```

### 3.1.3.3. Regresión lineal

El objetivo de un modelo de regresión lineal es tratar de explicar la relación que existe entre una variable dependiente (variable respuesta *Y*) y un conjunto de variables independientes (variables explicativas *X*). El algoritmo se ajusta a un modelo lineal para minimizar la suma de los cuadrados de las diferencias entre los valores reales/observados y los valores estimados por el modelo.

Se muestra una explicación gráfica en la Figura 3.5. Representación esquemática del algoritmo de Regresión Lineal .

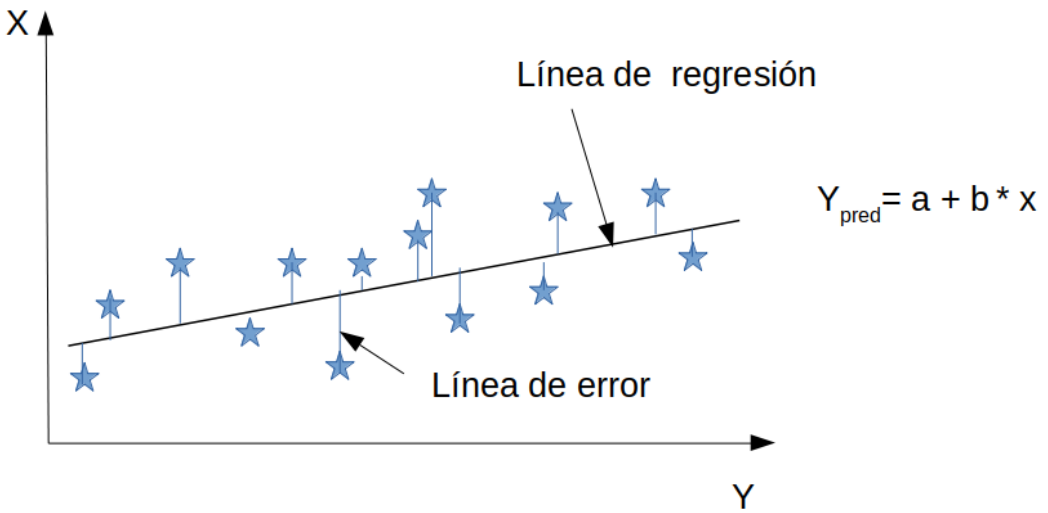


Figura 3.5. Representación esquemática del algoritmo de Regresión Lineal

Todos los algoritmos supervisados, como en el caso que nos ocupa, se requiere un set de datos etiquetado que se le presenta al algoritmo al mismo tiempo que le indica cuál es la respuesta correcta en cada caso. De esta forma el algoritmo puede "aprender" una función capaz de predecir el atributo objetivo para nuevas muestras (registros) de datos. Verificamos a través del siguiente test, que la clase lanza *NotOutputError* si no se proporciona el set etiquetado cuando se realiza el *fit* del modelo.

```
from aitenea.aitenea_core.linear_models.linear_regression import LRegression
from dask_glm.datasets import make_regression
from aitenea.exceptions.exceptions  import NotOutputError, NotFitError,ValidationError


def test_NotOutputRegr():
    """The test tried a linear regression algorithm. Launch a NotOutputError if target
       values is not given when fit the model
    """
    X, y = make_regression(n_samples=10, n_features=5)
    options = {"penalty":"l2", "tol": 0.0001,"solver": "admm"}
    parameters = {"options": options}
    lr = LRegression(parameters)
    try:
       lr.fit(X,y=None)
    except NotOutputError:
       assert True
```

### 3.1.4. Optimización de parámetros con algoritmia genética.

Los algoritmos de *machine learning* (3.1.3. Algoritmos de *machine learning*) tienen distintos hiperparámetros, que son parámetros ajustables que permiten controlar el proceso de entrenamiento de un modelo y de la optimización de los cuales va a depender, en gran medida, el rendimiento de dicho modelo. Los hiperparámetros, que pueden resultar óptimos, ante condiciones cambiantes en las entradas pueden dar peores resultados. Existen varios métodos de búsqueda de hiperparámetros: ajuste manual, búsqueda de cuadrícula (*grid search*), búsqueda aleatoria, optimización bayesiana, optimización basada en el gradiente u optimización evolutiva.

En el desarrollo incluido en nuestro *framework* hemos optado por implementar el método de optimización evolutiva, dado que consideramos que puede ser una buena alternativa en muchos casos, especialmente cuando la variabilidad de los hiperparámetros es grande. Una ventaja de este método es que nos proporciona una selección de los "más aptos" de tal manera que los mejores calificados en la carrera evolutiva pueden usarse de forma colaborativa, como segundas opciones o trabajando para entornos específicos, cada uno donde haya dado los mejores resultados. Un algoritmo genético tiene como objetivo

optimizar una función que depende de los hiperparámetros definidos. La cadena evolutiva termina cuando se cumple un criterio de finalización, que puede ser: un valor mínimo para la función de optimización, un número total de generaciones o un criterio de convergencia.

El proceso que sigue un algoritmo evolutivo se representa esquemáticamente en la Figura 3.6. Representación esquemática del algoritmo evolutivo. Se parte de una generación de individuos (generación 0), se somete a reproducciones y mutaciones de los individuos, y posteriormente se evalúa la función *fitness*. Se repite el proceso hasta conseguir que se cumpla el criterio de convergencia.

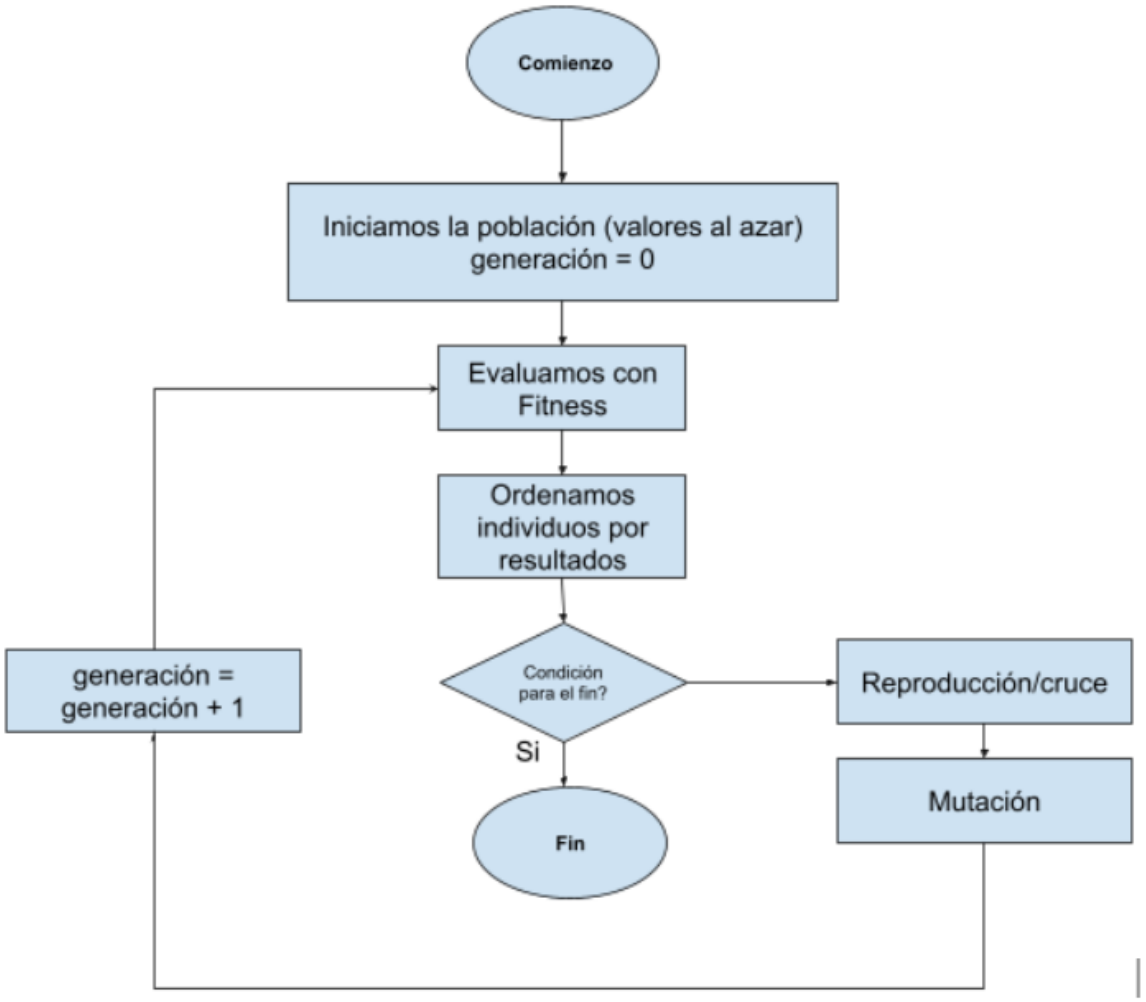



Figura 3.6. Representación esquemática del algoritmo evolutivo

Para que los hiperparámetros escogidos puedan funcionar como genes en el algoritmo evolutivo, es necesario realizar la codificación de los mismos. Inicialmente hemos investigado el método denominado *Dynamic Parameter Encoding* (DPE) por ser una solución clásica.

## 3.2. Panel de control de Altenea

Para nuestro proyecto AItenea ha sido desarrollado un panel de control con tres funciones: hace uso de la interfaz que pone a disposición la aplicación de Reports, recopila toda la documentación del proyecto y sirve como repositorio de las *Plines* (tuberías) creadas por un usuario, desde donde podrá explorarlas y, si lo desea, también administrarlas. En la Figura 3.7. Backend AItenea. Pantalla de login se muestra una captura de pantalla del *Control Center*.

El acceso al panel de control se realiza a través de la siguiente URL: http://0.0.0.0:7000/ o en su defecto los valores IP y puertos que el usuario haya escogido en el despliegue de AItenea.

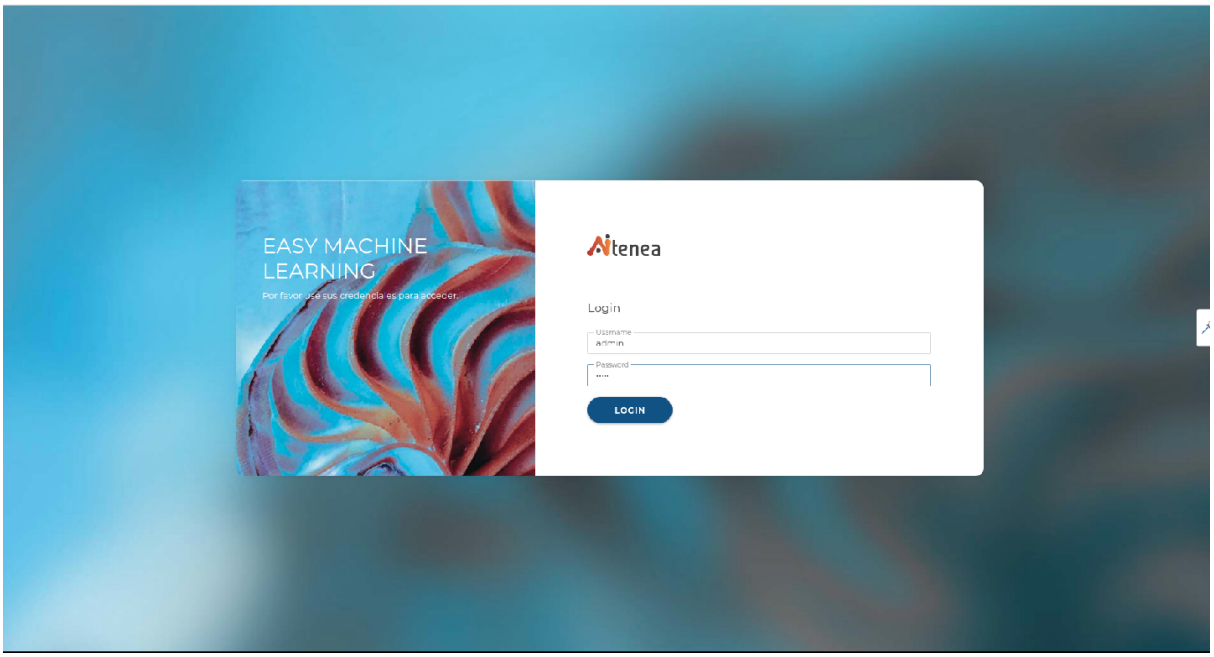



Figura 3.7. Backend AItenea. Pantalla de login

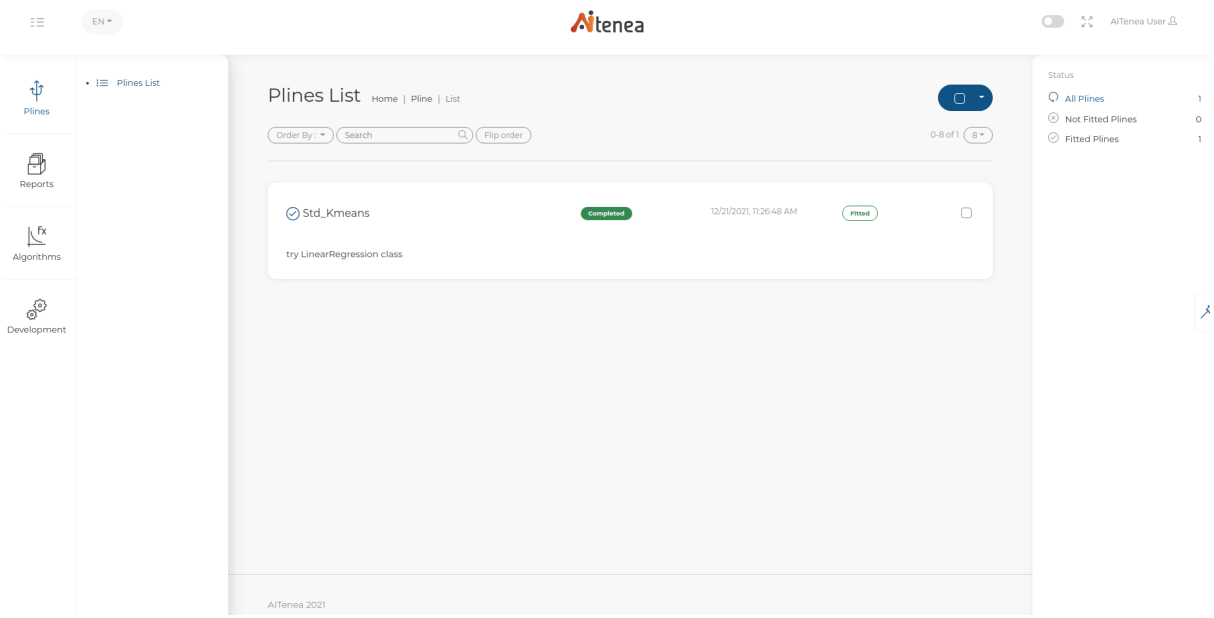



Figura 3.8. Backend AItenea. Pantalla Plines List

En la parte superior izquierda del panel el usuario puede escoger el idioma en el cual se le presenta la información: ES (Español) o EN (Inglés). El menú que se nos abre contiene en el panel izquierdo:

- Un listado de las tuberías: *Plines.*
- Un listado de reportes: *Reports.*
- Un listado de algoritmos: *Algorithms.*
- El panel de administración: *Development.*

### 3.2.1. Plines

Haciendo *click* en *Plines*, se nos ofrece una lista con las *Plines* (tuberías) Figura 3.9. Listado de las Plines creadas por el usuario. Se nos permite ordenar las *Plines* por el nombre o por la fecha de su creación.

La vista ofrece un filtro, en la parte derecha del panel, que nos permite seleccionar por las *Plines* totales, *Plines* entrenadas o *Plines* no entrenadas.

El botón azul ***Actions*** nos permite borrar las *Plines,* una por una, o todas a la vez.

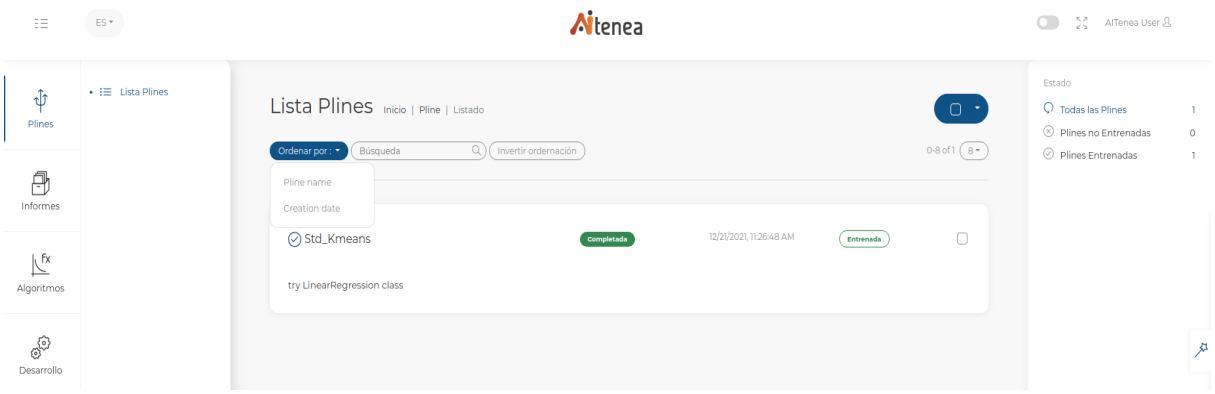



Figura 3.9. Listado de las Plines

También se puede elegir el número de resultados que se muestran en la página: 4, 8, 12 o 20.

Haciendo *click* sobre una de las *Pline* de la lista, se despliega la siguiente información detallada Figura 3.10. Descripción de las Plines:

- El nombre de la *Pline*.
- Una descripción de la *Pline.*
- La fecha de creación de la misma.
- Información sobre la *Pline* (*show Metadata*).
- Los pasos que componen la *Pline*, con una descripción de cada uno de ellos:
  - ➢ Lista de opciones (*show step options*). Los parámetros del algoritmo que se haya utilizado en cada paso.
  - ➢ Lista de parámetros genéticos (*show genetic parameters*).

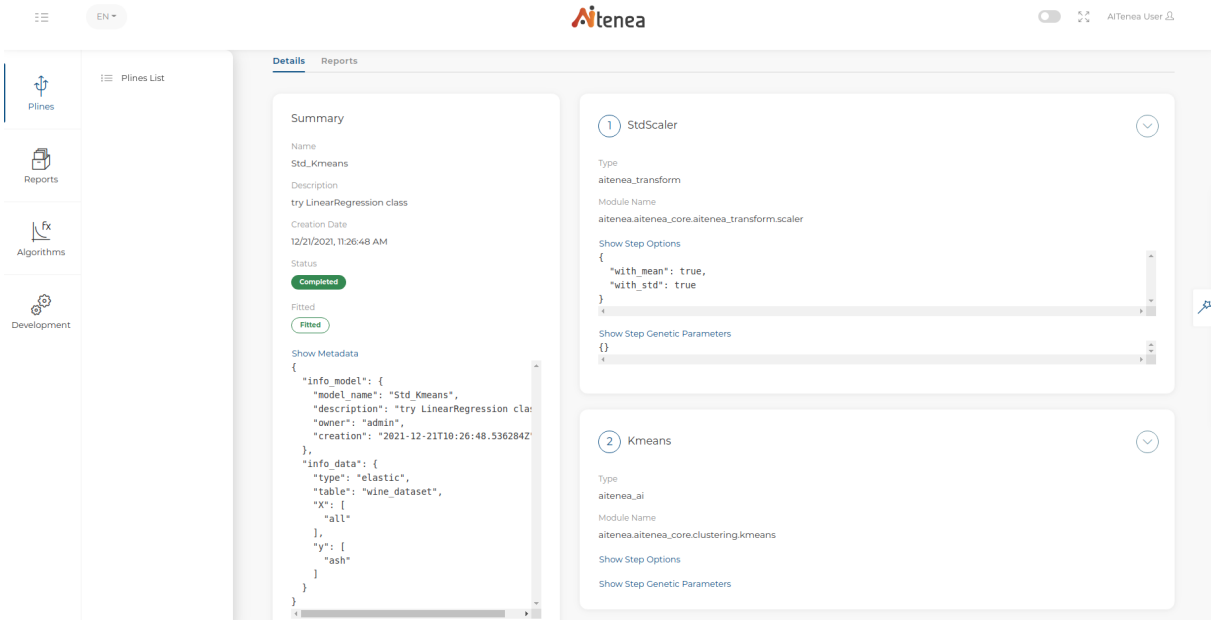


Figura 3.10. Descripción de las Plines

### 3.2.2. Reports

La función de este menú es gestionar el estado de la *Pline* durante su ejecución ofreciendo mensajes con información relevante sobre las mismas, y el control de errores. Estos reportes se despliegan a través del nodo *Redis-SUB* en la consola *Debug* de Node-RED.

Haciendo *click* en la pestaña *Reports* Figura 3.11. Descripción pestaña Reports se despliega información tal como: la fuente de datos, el porcentaje del set de datos que se ha usado para entrenamiento y para test, el *score* del modelo, listado de errores, etc..

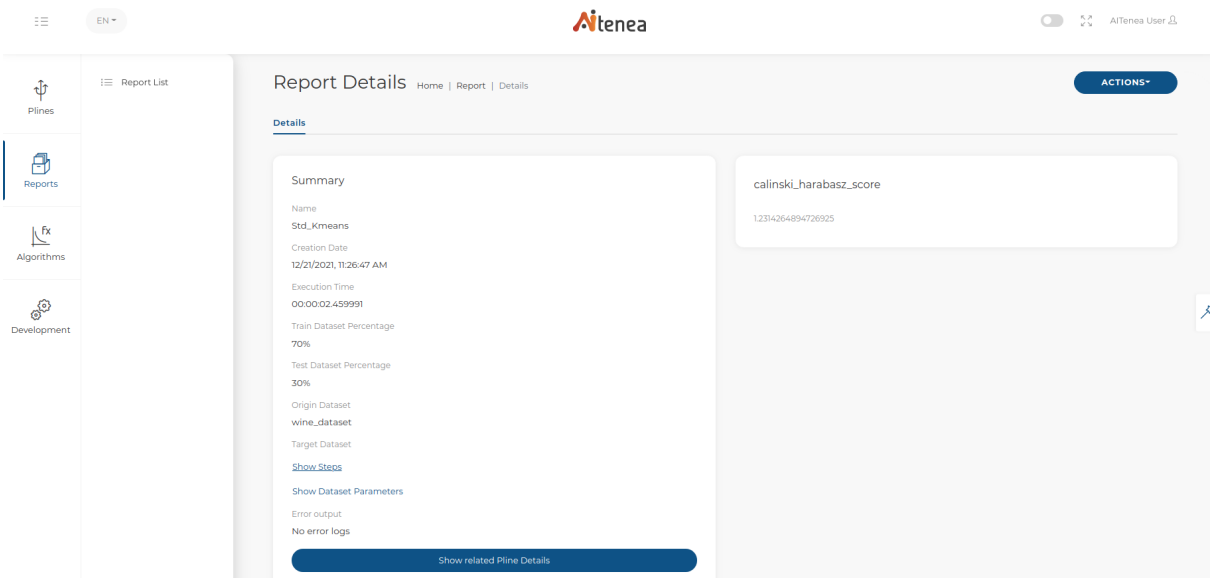



Figura 3.11. Descripción pestaña Reports

### 3.2.3. Algorithms

Haciendo *click* en la pestaña *Algorithms* Figura 3.12. Pantalla de lista de algoritmos se muestra una lista con los algoritmos implementados en AItenea. Se pueden ordenar según su tipología o según el nombre, y filtrar una u otra categoría, o todos (véase panel derecho).

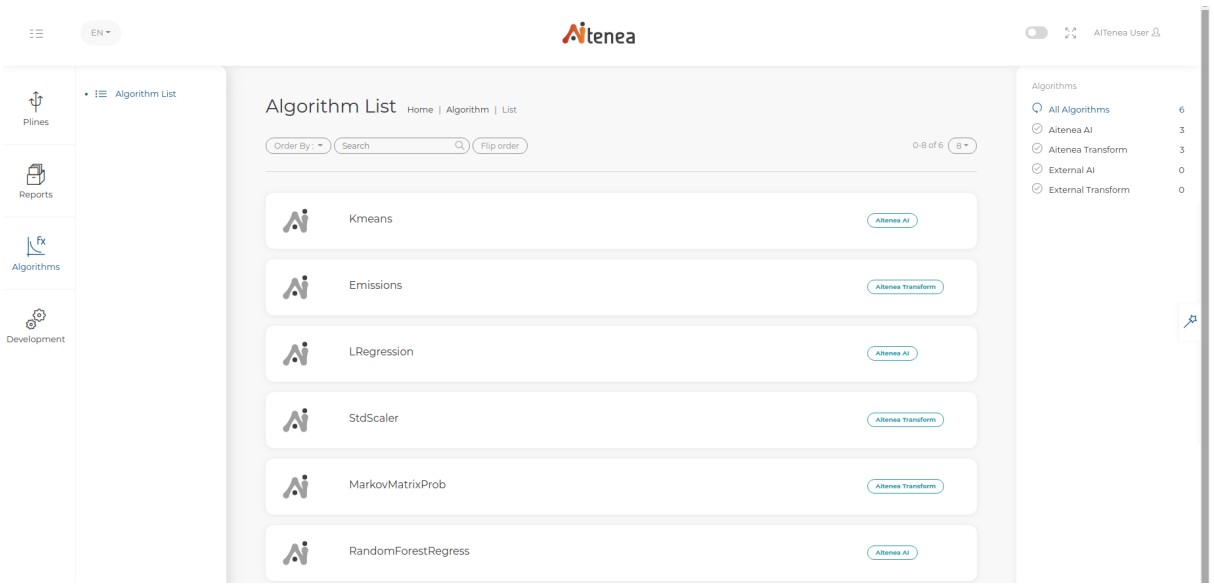



Figura 3.12. Pantalla de lista de algoritmos

Haciendo *click* sobre un algoritmo de la lista se ofrece información como: el nombre del mismo, tipología, parámetros del algoritmo, parámetros genéticos, etc. También ofrece información sobre la fuente de datos y las variables (*X*) y objetivo (*y*) seleccionadas Figura 3.13. Pantalla de detalle de un algoritmo.

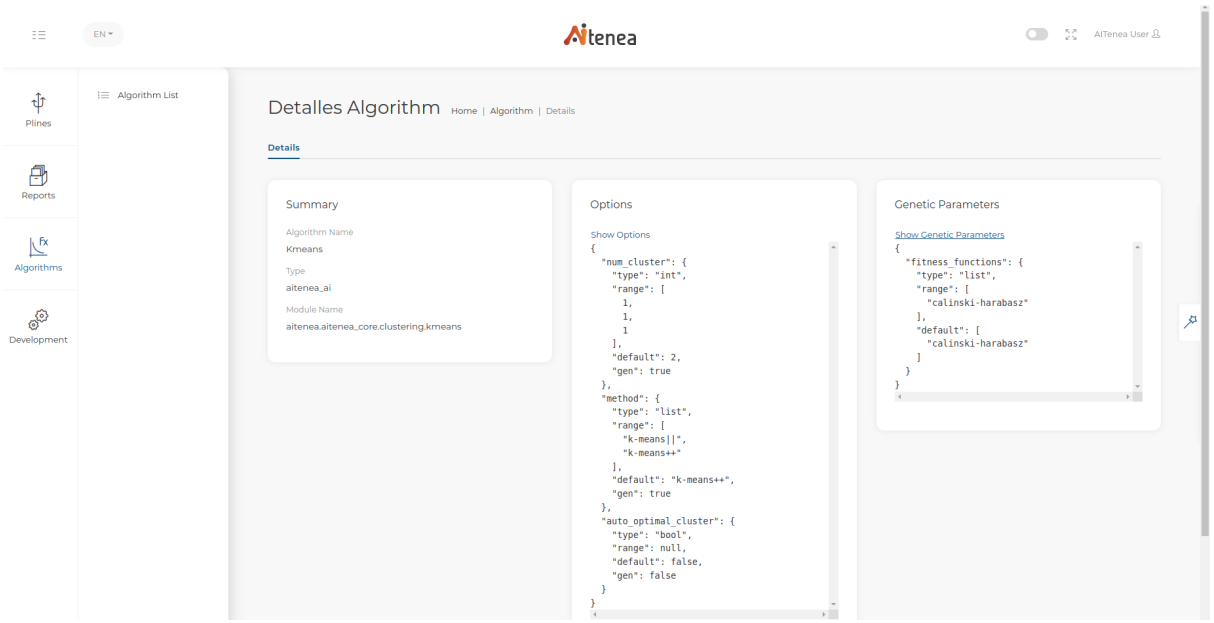



Figura 3.13. Pantalla de detalle de un algoritmo

### 3.2.4. Development

Haciendo *click* en la pestaña *Development* Figura 3.14. Descripción pestaña Development se despliegan las siguientes pestañas:

- Node-Red.
- Sphinx Docs.
- Sphinx Docs Index.
- *Redoc Docs*.
- API Swagger.

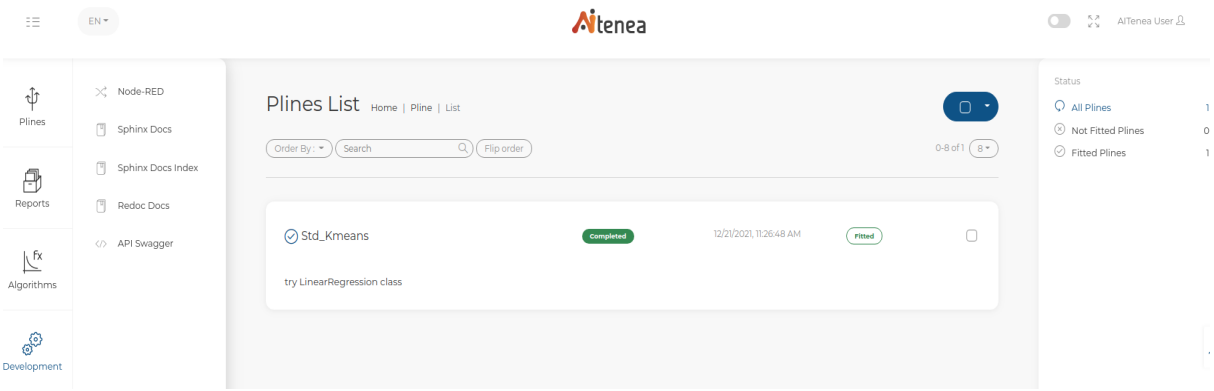



Figura 3.14. Descripción pestaña Development

Haciendo *click* sobre Node-Red se nos abre el acceso al AItenea Node-Desk 2.1. Un paseo por AItenea Node-Desk y el usuario puede realizar sus flujos de simulación y crear sus modelos, probar modelos ya creados y hacer sus predicciones, tal como se describe de forma detallada en el capítulo AItenea en 30 minutos.

Haciendo *click* sobre Sphinx Docs Figura 3.15. Vista proyecto AItenea se abre el acceso a la documentación de la herramienta AItenea, que incluye un manual del usuario, aitenea_core, aitenea_api, aitenea_reports y aitenea_nodes.


Figura 3.15. Vista proyecto AItenea

Haciendo *click* sobre Sphinx Docs Index Figura 3.16. Listado índices AItenea se nos abre el listado de índices AItenea.

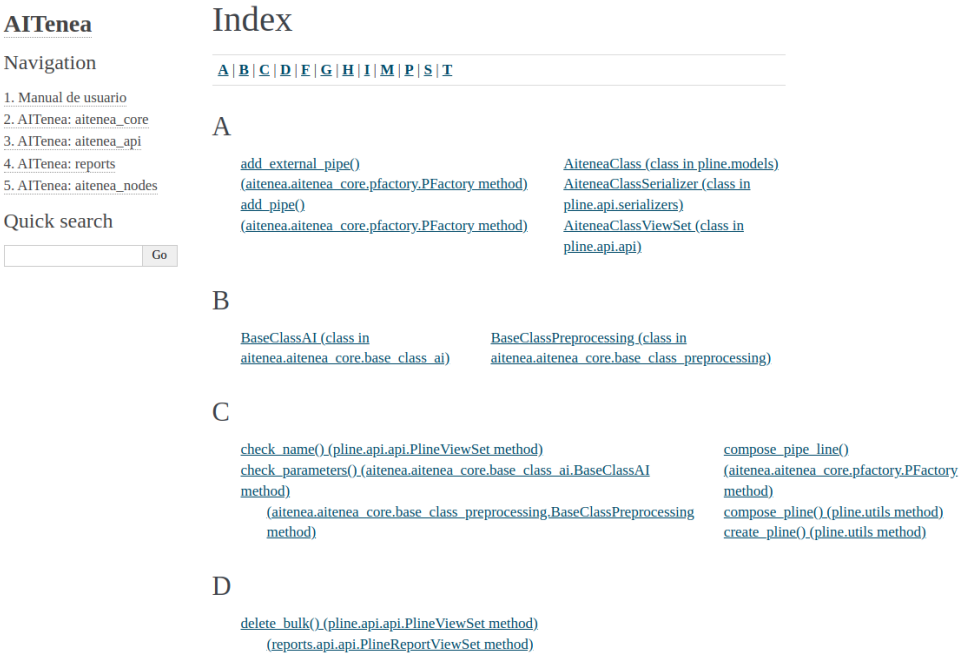



Figura 3.16. Listado índices AItenea

Haciendo *click* sobre Redoc Docs Figura 3.17. Documentación Redoc Docs, se nos abre acceso a la documentación autogenerada por la API.

Ofrece documentación de cada clase, con ejemplos de peticiones y respuestas en el panel derecho.

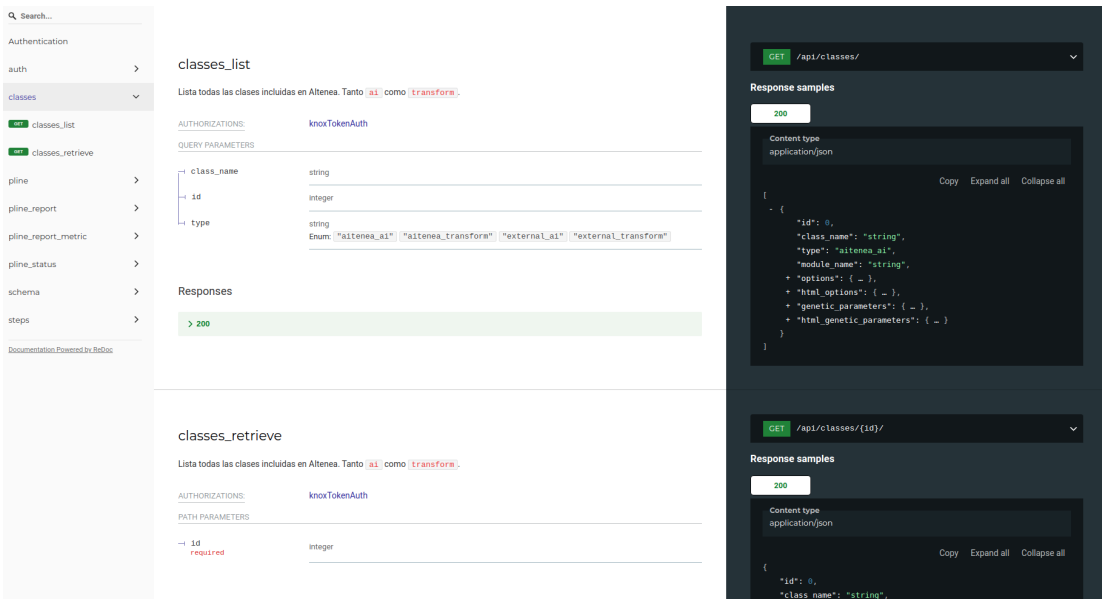



Figura 3.17. Documentación Redoc Docs

Haciendo *click* sobre API Swagger Figura 3.18. Gestor de llamadas API Swagger se nos abre una interfaz, como alternativa a un gestor de llamadas a la API (*Postman*). Este es muy útil para probar diferentes peticiones.

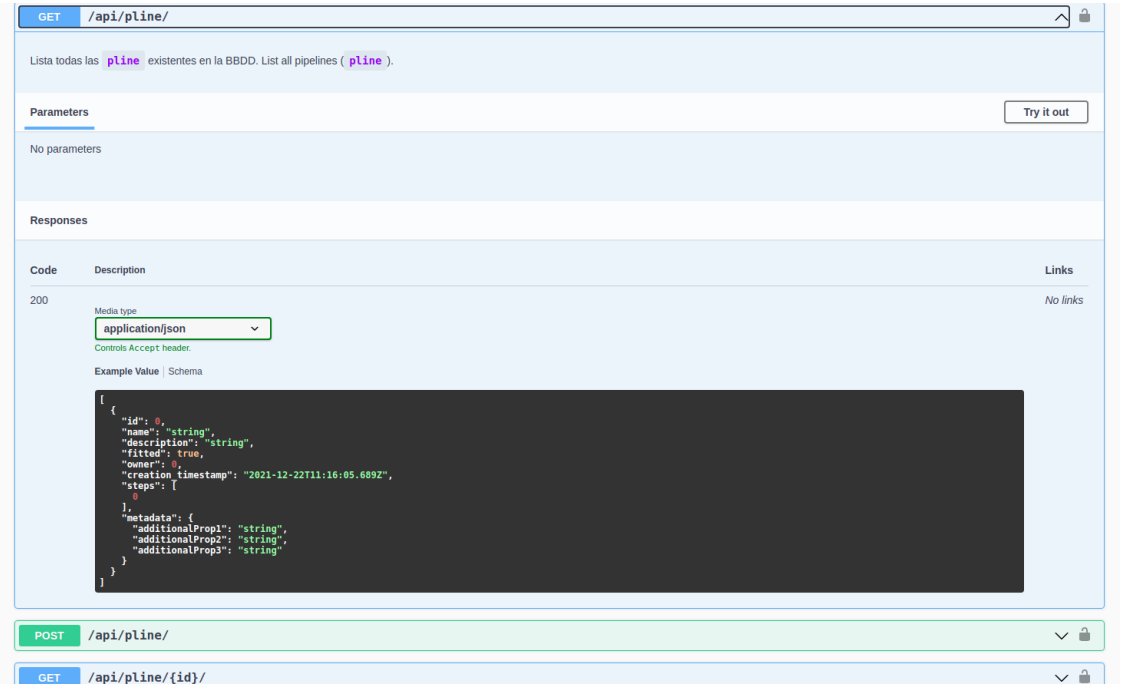



Figura 3.18. Gestor de llamadas API Swagger

## 3.3. Capa de alto nivel

### 3.3.1. Estructura de directorios de aitenea_node-red

Mostramos a continuación una representación esquemática, simplificada, de la estructura de directorios de *aitenea-node-red*:

*|---- aitenea_node-red*

*|---- node_red_data*

*|---- aitenea_nodes*

*|---- aitenea-conf*

*aitenea_conf.html*

*aitenea_conf.js*

*package.json*

*|---- data-csv*

*|---- data-elasticsearch*

*|---- data-evaluation*

*|---- destination-elastic*

*|---- elastic-conf*

*|---- manual-data*

*|---- model-get*

*|---- pline-run*

*|---- pline-set*

*|---- pline-step*

*|---- redis-sub*

La arquitectura de los nodos sigue, tanto para los de AItenea como para el resto de elementos de Node-Red el siguiente esquema:

Nodo-Aitenea:

- Un archivo .js del que define el comportamiento en tiempo de ejecución del nodo.
- Un archivo .html que define cómo aparece el nodo con el editor. Contiene tres partes distintas, cada una envuelta en su propia etiqueta <script>:
    - La definición del nodo principal registrada en el editor. Esto define la categoría a la que pertenece, las propiedades editables (valores predeterminados), el icono que usa, etc.. Está dentro de una etiqueta de script JavaScript.
    - La plantilla de edición define el contenido del cuadro de diálogo de edición para el nodo. Se define en un script de tipo text / html.
    - El texto de ayuda que se muestra en la pestaña de la barra lateral de información. Se define en un script de tipo text / html.
- Un archivo package.json que es un estándar utilizado por los módulos de Node.js para describir su contenido.

En el esquema arriba mencionado cada uno de los directorios que se encuentran en *aitenea_nodes* contiene los tres tipos de fichero arriba mencionados, que definen la arquitectura del mismo, tal como se ha ejemplificado en el nodo *aitenea_conf*.

Los nodos específicos implementados en AItenea para cubrir con los objetivos de nuestro desarrollo han sido agrupados en las siguientes tres categorías, *Data Cleaning*, *Connector* y *Smart Pipe*, según la funcionalidad que desempeñan y que describimos a continuación.

### 3.3.2. Nodos Data Cleaning

A través del nodo *Data-evaluation* el usuario carga un fichero y realiza una estadística básica de los datos obteniendo el valor mínimo, máximo, los percentiles, la media, la varianza, tal como se muestra en la Figura 3.19. Nodo Data Evaluation. Es muy útil para hacer un estudio previo básico sobre los datos con los que vayamos a trabajar.

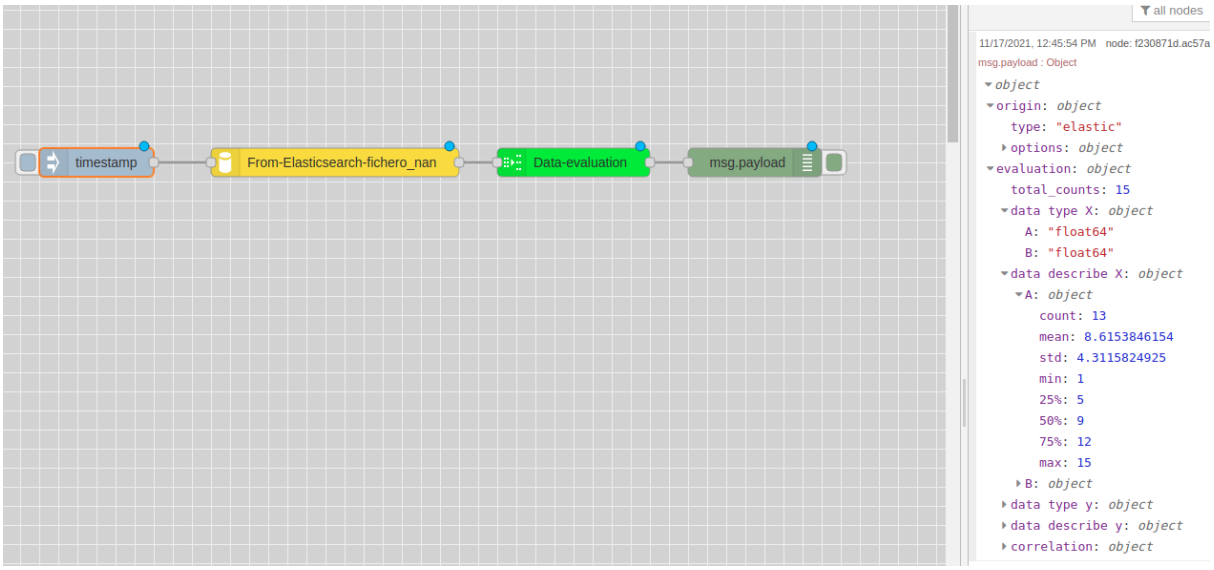



Figura 3.19. Nodo Data Evaluation

### 3.3.3. Nodos Connector

El usuario dispone de una serie de nodos cuya funcionalidad es de gestionar índices en la base de datos, *From-ElasticSearch*, *Destination-Elastic*, o generar un set de datos artificial con *Manual-data*.

El nodo *From-Elasticsearch* Figura 3.20. Nodo From Elasticsearch, identificación de usuario… tiene la funcionalidad de acceder a los índices existentes en una base de datos Elasticsearch, a la que el usuario tenga acceso. Al desplegar el nodo se abre la siguiente ventana, donde se nos piden los datos de conexión a la base de datos (pulsando el símbolo lápiz se abre la ventana de conexión de la base de datos) y nombrar el nodo en la opción *Node Name*.

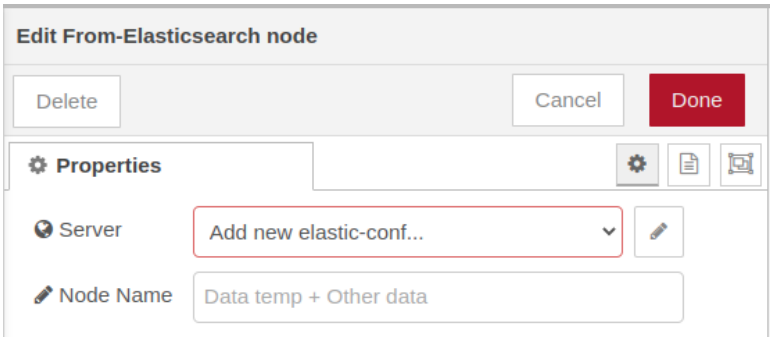



Figura 3.20. Nodo From Elasticsearch, identificación de usuario

Posteriormente se abre una ventana en la que se requiere introducir la siguiente información Figura 3.21. Nodo From Elasticsearch:

- Escoger un índice.
- Seleccionar los atributos *X*.
- Seleccionar un valor objetivo *Y*, si el modelo lo requiere.
- Se ofrece la posibilidad de seleccionar los datos a través de una *query*, tal como se muestra en el siguiente código como ejemplo.

```
{
"bool": {
  "must": [],
  "filter": [
    {
      "match_all": {}
    }
  ],
  "should": [],
  "must_not": [
    {
      "match_phrase": {
        "C": "5"
      }
    }
  ]
}
```

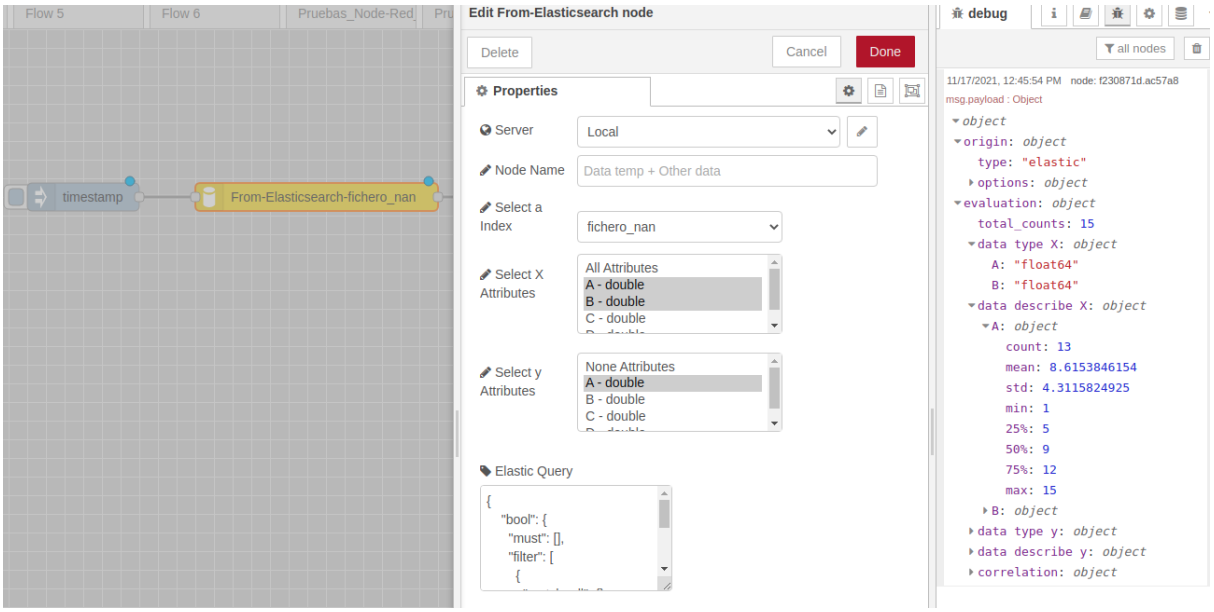



Figura 3.21. Nodo From Elasticsearch

Se confirma la configuración pulsando en *Done*.

El nodo *Destination-Elastic* Figura 3.22. Nodo Destination Elastic tiene la funcionalidad de modificar índices existentes o crear nuevos índices. Al desplegar el nodo se abre la siguiente ventana, donde se nos piden los datos de conexión a la base de datos (pulsando el símbolo lápiz se abre la ventana de conexión de la base de datos). Se pide posteriormente la siguiente información:

- Nombrar el índice.
- Existe la posibilidad de seleccionar que se sobreescriba el índice en caso de que ya exista, si el usuario así lo considera.

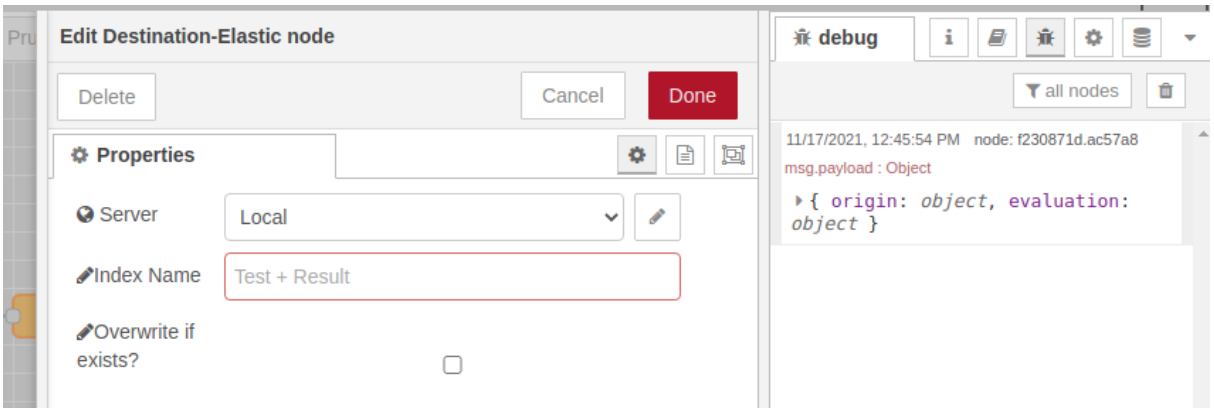



Figura 3.22. Nodo Destination Elastic

El despliegue del nodo *Manual-data* Figura 3.23. Nodo Manual-data abre una ventana, como la que se muestra en las siguientes figuras, que permite introducir un set de datos artificial en formato JSON Figura 3.24. Nodo Manual-data - Edit JSON.

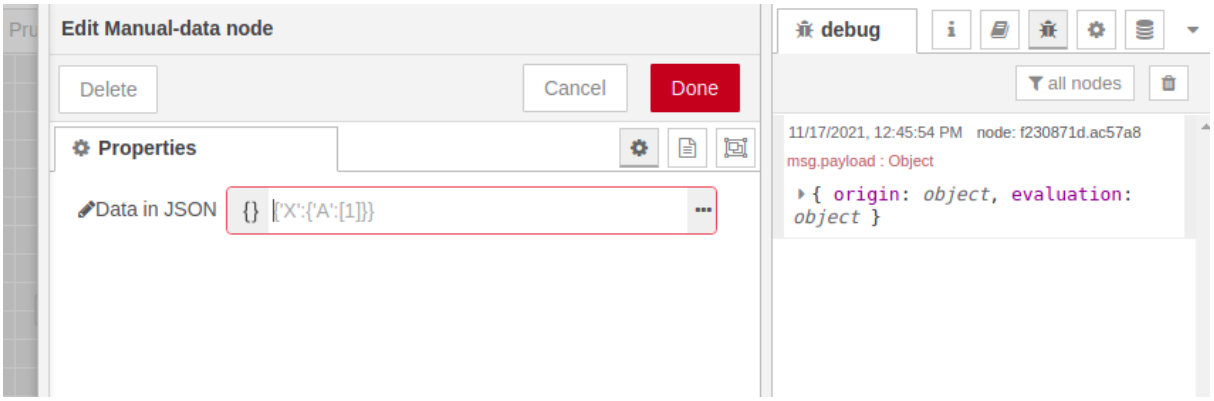



Figura 3.23. Nodo Manual-data

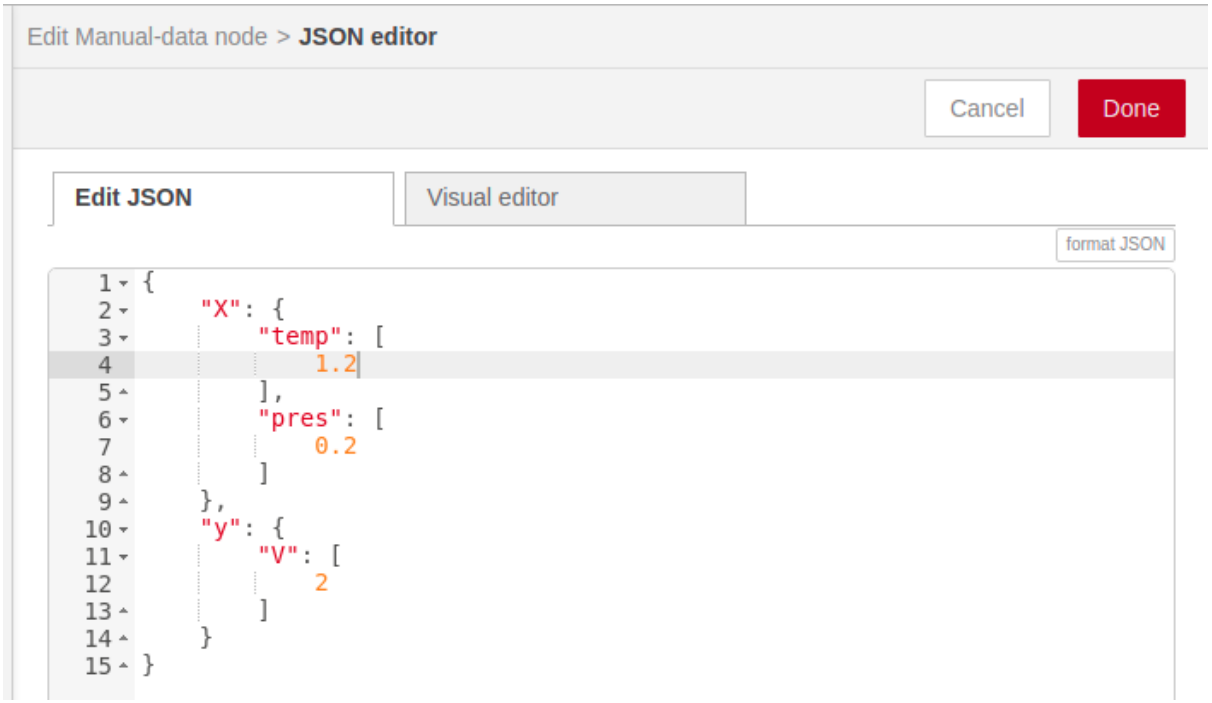



Figura 3.24. Nodo Manual-data - Edit JSON

La funcionalidad del nodo *From*-CSV Figura 3.25. Nodo From CSV autentificación. es la de gestionar datos desde un CSV, y posteriormente usarlos en los algoritmos implementados en AItenea. En su despliegue se requiere de los siguientes datos por parte del usuario :

- El nombre del usuario de AItenea.
- La clave del usuario de AItenea.

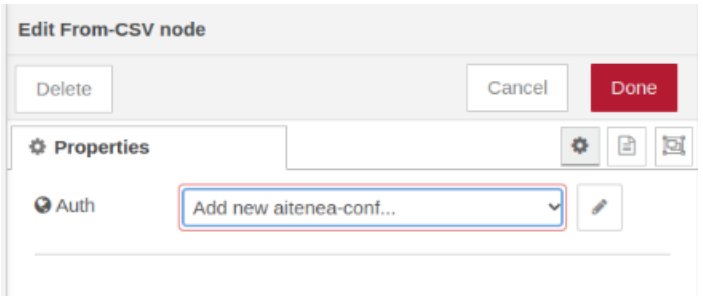


Figura 3.25. Nodo From CSV autentificación.

Una vez introducidos los datos se guarda la configuración haciendo *click* en *Done*, y se abre una nueva ventana donde el usuario tiene la opción de elegir un índice, en la opción *Select a Index,* entre los ya existentes y se le muestra un desplegable con las variables de ese índice que el usuario podrá gestionar Figura 3.26. Nodo From CSV. Acceso a un….

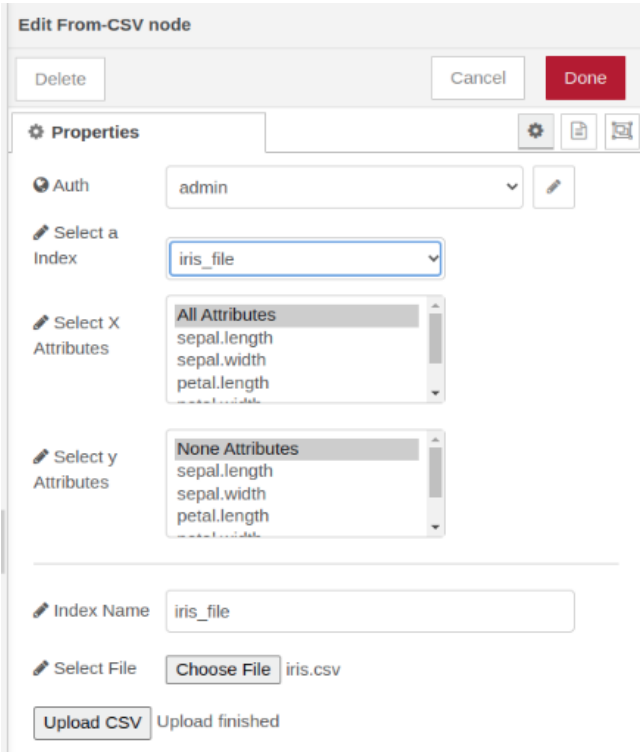



Figura 3.26. Nodo From CSV. Acceso a un índice creado previamente.

Además, el usuario cuenta con la opción de subir un archivo CSV como un nuevo índice siguiendo los siguientes requisitos Figura 3.27. Nodo From CSV. Subir fichero CSV:

- Se debe dar un nombre al nuevo índice en la opción *Index Name*, los caracteres permitidos son letras, número, guiones y guiones bajos.
- El usuario debe pulsar en *Choose File* y elegir archivo de su equipo, este debe contar con una cabecera y tener un delimitador estándar, por ejemplo, comas o espacios.
- Una vez se pulsa en "*Upload CSV*" se mostrará un texto de estado que indicará si la subida sigue en curso o ha finalizado, si ocurre un error, se informará con un mensaje de alerta.

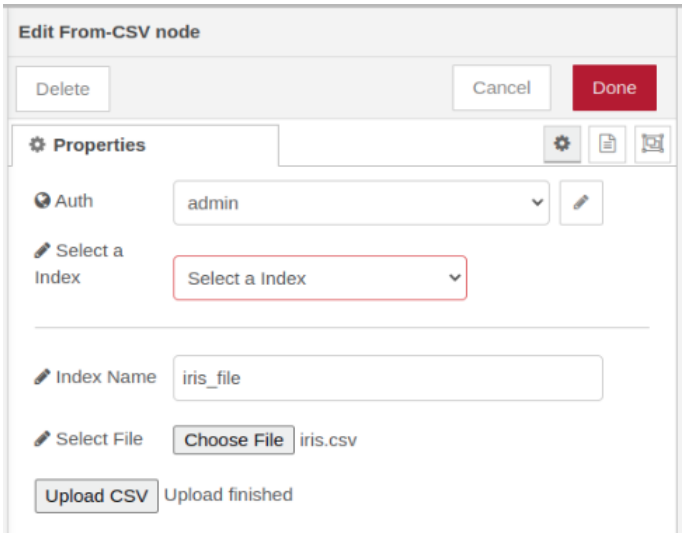



Figura 3.27. Nodo From CSV. Subir fichero CSV

### 3.3.4. Smart Pipe

Esta categoría agrupa los nodos cuya funcionalidad es la de crear tuberías, añadir tareas a las mismas, y ejecutar el modelo definido.

Tal como hemos mencionado anteriormente, una tubería es un flujo completo de funcionalidad que puede realizar cualquiera de las siguientes acciones:

1. Transformar datos según una o varias transformaciones realizadas en serie.
2. Transformar los datos y aplicar finalmente un modelo para entrenamiento.
3. Aplicar un modelo para entrenamiento sin transformar los datos.

En el nodo *Pline-set*, como primer paso se nos pide nuestros datos de usuario Figura 3.28. Nodo Pline-set, identificación de…. Pulsando sobre el símbolo lápiz se puede acceder a los datos de conexión a nuestra base de datos. Seguidamente se nombra la tubería y se introduce una pequeña descripción de la misma Figura 3.29. Nodo Pline-set. Introducción de….

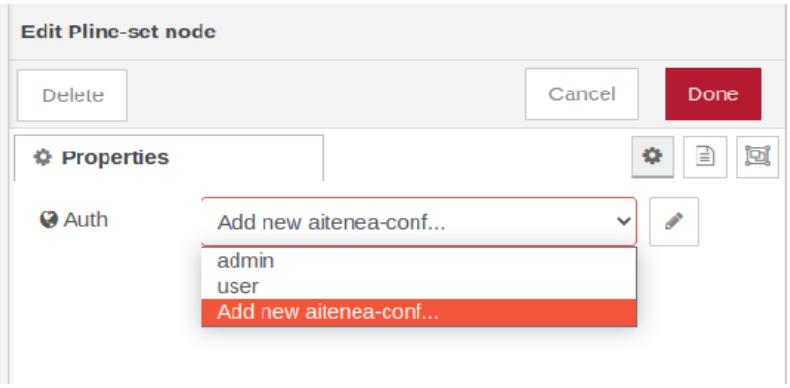


Figura 3.28. Nodo Pline-set, identificación de usuario

Obsérvese que existe la posibilidad de re-escribir la tubería, lo que evidentemente elimina la consecución de acciones de la anterior tubería.

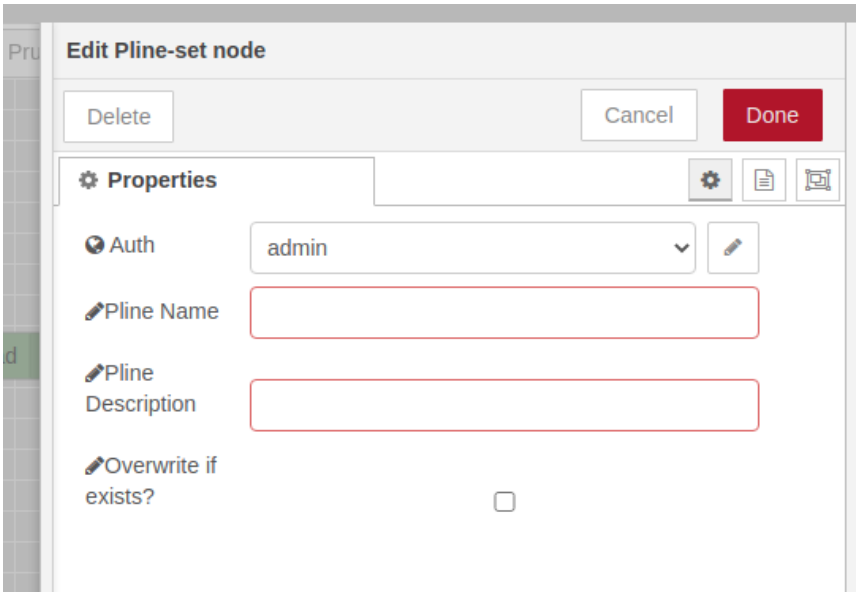



Figura 3.29. Nodo Pline-set. Introducción de datos pline

El propósito de la tubería es agrupar varios pasos (tareas) que se pueden validar de forma cruzada juntos mientras se establecen diferentes parámetros para cada uno de ellos.

En el despliegue del nodo *Pline-step* Figura 3.30. Nodo Pline-step, identificación de…, cuya funcionalidad es de introducir sucesivamente tareas a la tubería previamente creada, se nos pide como primer paso acceder con nuestro usuario.

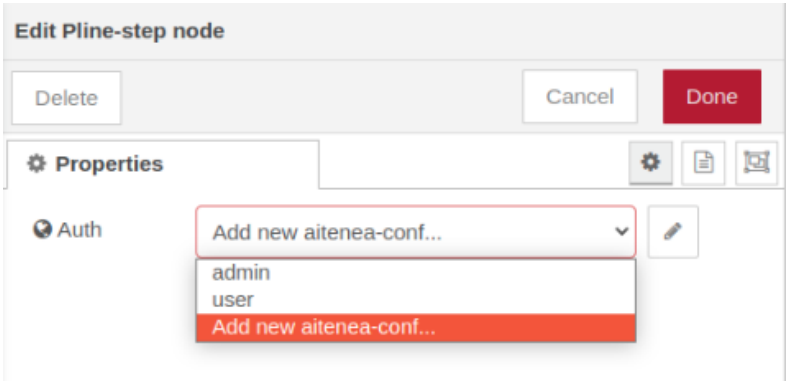



Figura 3.30. Nodo Pline-step, identificación de usuario

Seguidamente se nos abrirá una lista con todas las tareas implementadas en AItenea, que pueden ser de transformación (*aitenea_transform*) o de *machine learning* (*aitenea_ai*), en la opción *Select Class Type*, y se nos despliega la lista de algoritmos de tipo de la clase seleccionado Figura 3.31. Nodo Pline-step. Selección de….

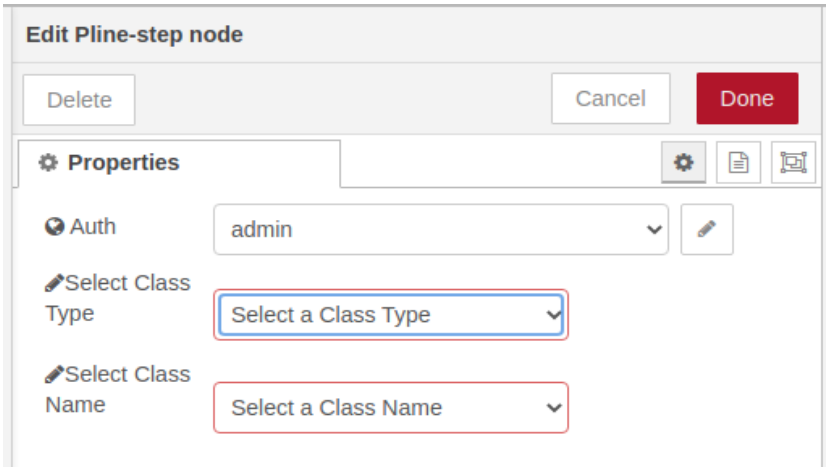



Figura 3.31. Nodo Pline-step. Selección de tipo de clase y clase

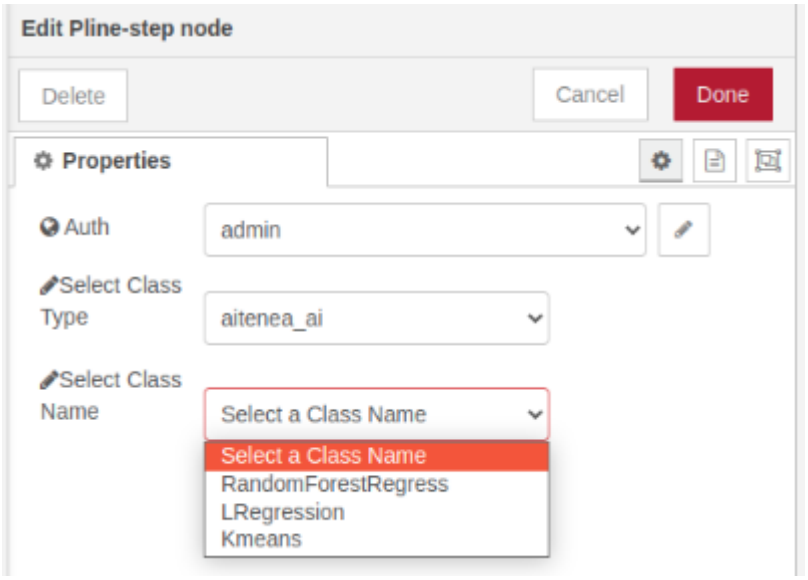



Figura 3.32. Nodo Pline-step. Detalle de clases de ai

En el despliegue del nodo *Pline-run* Figura 3.33. Nodo Pline-run, cuya funcionalidad es de 'ejecutar' la tubería, se nos requerirá:

- Seleccionar una acción de la lista de acciones disponibles: *fit*, *fit_transform*, *fit_predict*, *predict*.
- Seleccionar el porcentaje del set de datos que se usará para el entrenamiento.

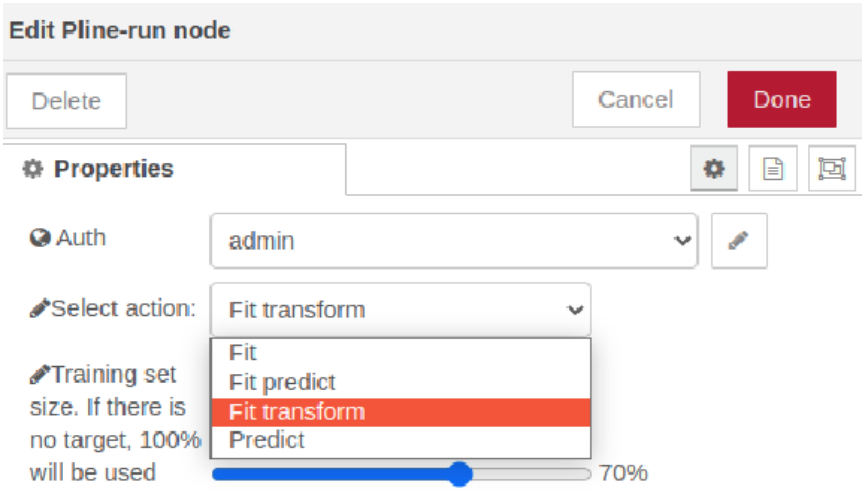



Figura 3.33. Nodo Pline-run

Una vez que se ha creado un modelo en AItenea, posteriormente se puede cargar el modelo y usar con nuevos datos para su transformación, o para hacer predicciones. Todo esto a través del nodo *Model-get* Figura 3.34. Nodo Model-get, identificación de…En la primera ventana de despliegue se nos pide nuestra autentificación.

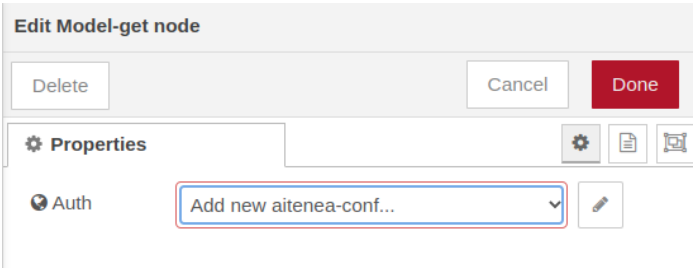



Figura 3.34. Nodo Model-get, identificación de usuario

Posteriormente se nos abre la ventana de selección del modelo (*Select pLine Name*), entre los ya existentes en AItenea Figura 3.35. Nodo Model-get.

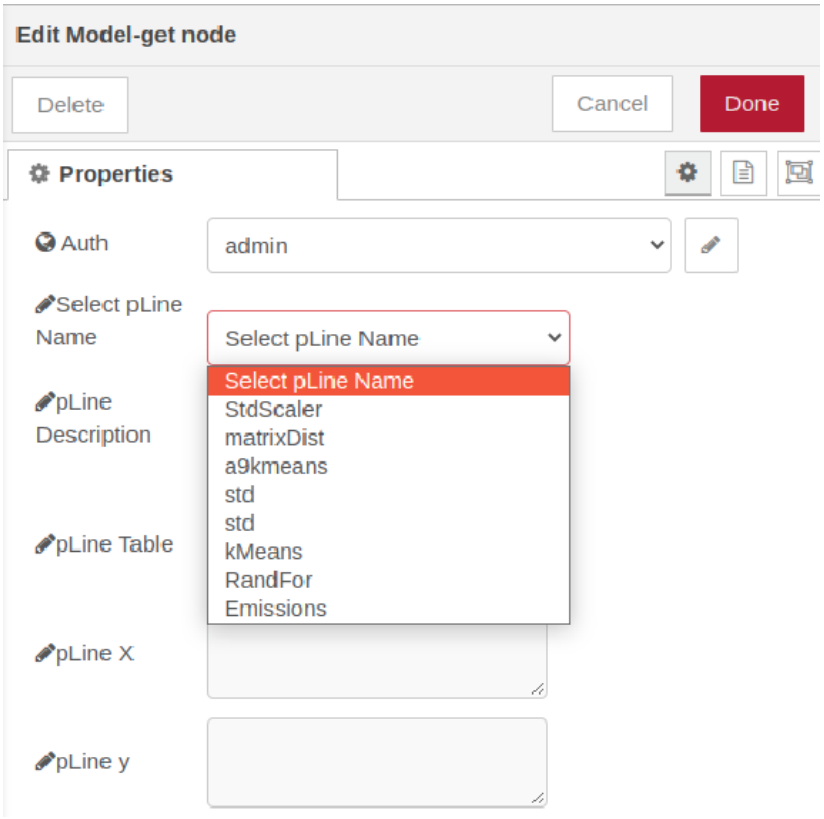



Figura 3.35. Nodo Model-get, selección del modelo

Una vez seleccionado un modelo, de forma automática se nos despliega toda la información del mismo: la fuente de datos, las variables predictoras (*pLine X*) y objetivo (*pLine y*) que se han usado al crear el modelo Figura 3.36. Nodo Model-get, información del….

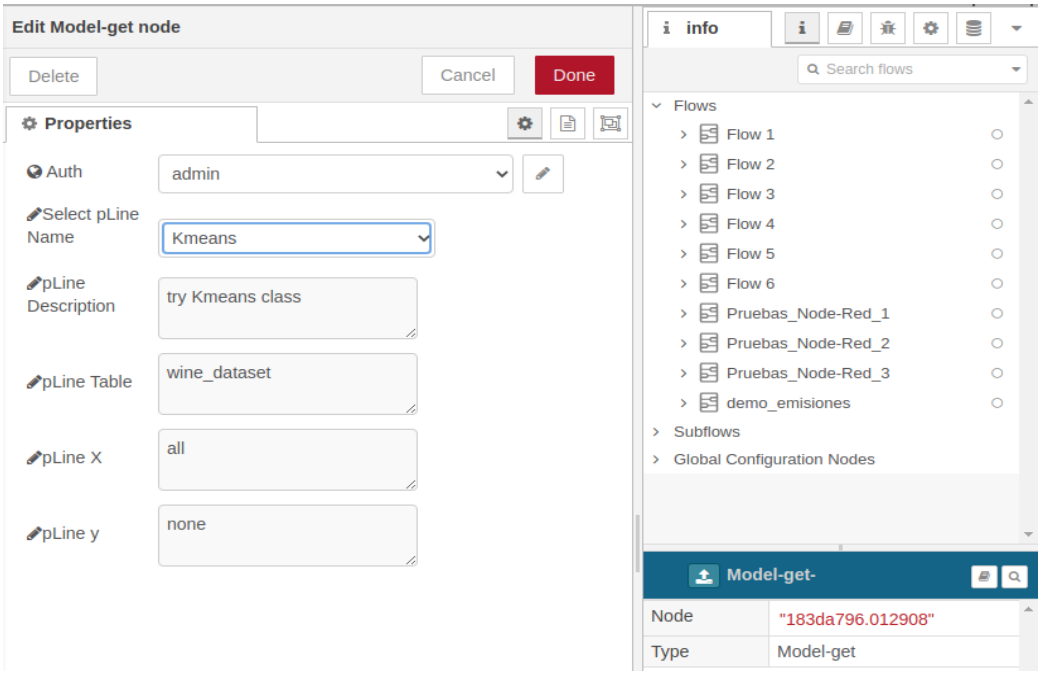



Figura 3.36. Nodo Model-get, información del modelo

En el despliegue del nodo *Redis-SUB* Figura 3.37. Nodo Redis-SUB. Configuración, cuya funcionalidad es de actualizar el estado de ejecución de la *Pline* mediante un indicador visual y *logs* en la consola de *Node-RED*.

- Deberemos introducir una cadena de conexión en el nodo, debe de ser la misma que use AItenea si queremos mantener la funcionalidad en el backend.

Ejemplo de cadena de conexión:

```
{
   "host":"0.0.0.0",
   "port":6379,
   "password":"password",
   "db":0
}
```

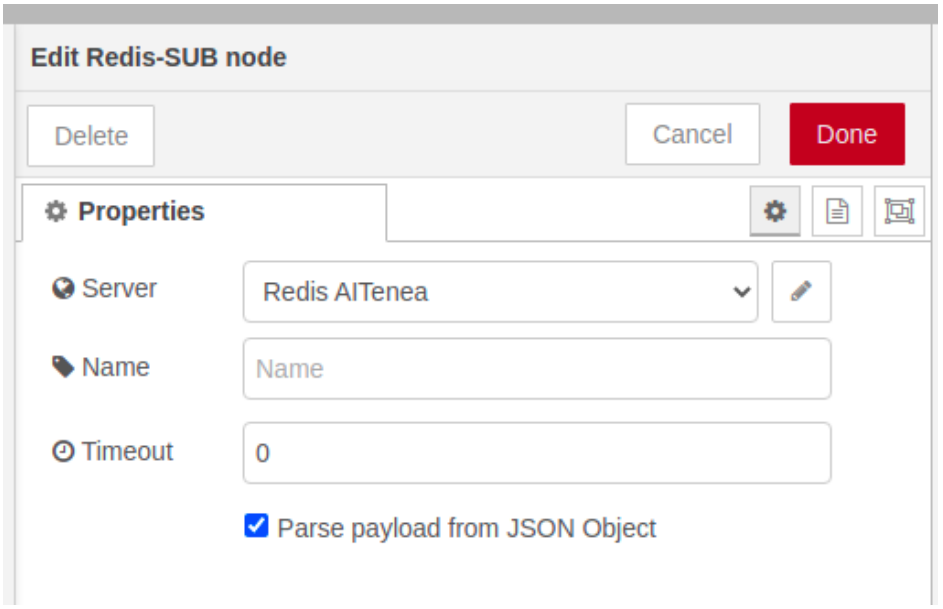



Figura 3.37. Nodo Redis-SUB. Configuración

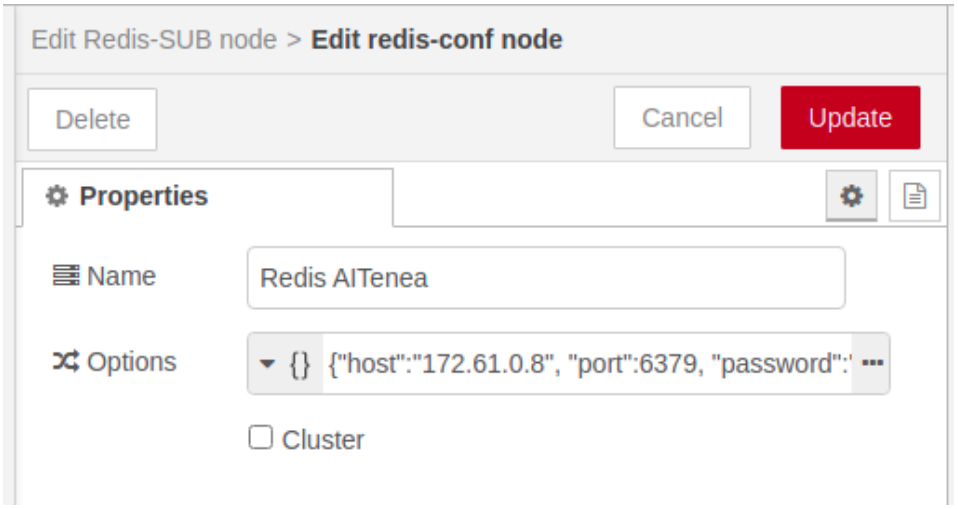



Figura 3.38. Nodo Redis-SUB. Opciones

# 4. Guía de desarrollo

Este capítulo desea ser una guía, a nivel de desarrollador, a la hora de implementar nuevos algoritmos (de transformación o de *machine learning*) en la capa de bajo nivel, o para la implementación de nuevos nodos con la funcionalidad deseada en la capa de alto nivel.

## 4.1. Implementación de algoritmos

Describimos la metodología a seguir, a nivel de desarrollador, a la hora de implementar nuevos algoritmos en AItenea. Tal como se ha explicado en detalle en 3.1. Capa de bajo nivel esta implementación se hace en la capa de bajo nivel, *aitenea_core*.

El desarrollo de las clases presentes en *aitenea_core* se fundamenta básicamente sobre dos clases base, *base_class_preprocessing,* de la que heredan las clases de transformación y *base_class_ai,* de la que heredan clases cuya función es la de desarrollar algoritmos de *machine learning*.

A la hora de implementar una nueva funcionalidad en *aitenea_core* se debe seguir el siguiente esquema de directorios:

*nueva_funcionalidad/*
- *__init__.py*
- *class_name.py*
- *_auxiliar_files.py*
- *test/*

Si ya existe un directorio con algoritmos cuya tipología fuese similar al que se desea implementar, se puede añadir en el mismo directorio el fichero *class_name.py*, y la clase auxiliar si así se considera 4.1.4.2. Clases auxiliares.

Siguiendo la guía de estilos PEP 8 destacamos los siguientes aspectos:

- Los nombres de los módulos deben ir en minúsculas y guión bajo.
- Los nombres de las clases deben ir en mayúsculas y sin guión bajo.
- Los nombres de las variables deben ser términos cortos, representativos y en minúsculas.
- En el método *__init__* se deben definir todos los atributos de la clase aunque se inicien en *None*.

### 4.1.1. Implementación de algoritmos de transformación

Según requerido por la clase base, los parámetros de la clase de transformación implementada, se deben introducir en forma de diccionario. Para cada parámetro se debe proporcionar: *type* el tipo del mismo (*int, str*,..etc), *default* un valor por defecto, y una variable *gen* que si toma valor *True* el parámetro se usa como gen en un algoritmo evolutivo, tal como se muestra en el siguiente ejemplo:

```
class_options = {"parameter1":{"type":"  ","default":"  ","gen":"  "},

                "parameter2":{"type":"  ","default":"  ","gen":"  "},};

self.class_parameters = {"options":class_options};
```

Los parámetros son los elegidos por el programador y deben de ser representativos del algoritmo implementado.

Cualquier clase de transformación que se desee implementar debe de contener, según requerido por la clase base, los siguientes métodos: *get_info*, *init_selector*, *fit*, *transform*, *fit_transform*.

El método *get_info* debe ofrecer información general sobre el algoritmo implementado: atributos, objetivo, funcionalidad.

El método *init_selector* es el método en el cual se realiza la inicialización del algoritmo, en caso de que sea necesario.

El método *fit* es implementado por cada estimador, acepta como entrada de datos los atributos *X* del modelo y la variable objetivo *Y*, para los modelos supervisados. Normalmente, el estimador realiza la validación de parámetros y los datos. También es responsable de estimar los atributos de los datos de entrada y almacenar los atributos del modelo y finalmente devolver el resultado a través del parámetro *self*.

El método *transform* acepta como datos de entrada los atributos *X*. Los parámetros generados a partir del método *fit*, aplicados sobre el modelo devuelven un conjunto de datos transformados.

El método *fit_transform* aplica en un mismo paso el *fit* y posteriormente *transform*. Recibe como datos de entrada los atributos *X* y la variable objetivo *Y*, y devuelve como salida el set de datos transformado.

### 4.1.2. Implementación de algoritmos de machine learning

Según requerido por la clase base, los parámetros de la clase de *machine learning* implementada se deben introducir en forma de diccionario, tal como se muestra en el siguiente ejemplo:

*class_options={"parameter1":{"type":" ","range":" ", "default":" ","gen":" "},*

*"parameter2":{"type":" ","range":" ", "default":" ","gen":" "},};*

*self.class_parameters ={"options":class_options};*

*class_genetic_parameters = {"fitness_functions": {"type": " ", "range": [" "], "default": [" "]}};*

*self.class_genetic_parameters = {"options": class_genetic_parameters};*

Los parámetros son los elegidos por el programador como representativos del algoritmo implementado. Si el modelo implementado se usa dentro de un algoritmo genético se debe proporcionar la función *fitness* que determinará la selección de los mejores individuos que seguirán evolucionando.

Cualquier clase de *machine learning* que se desee implementar debe de contener, según requerido por la clase base, los siguientes métodos: *get_info*, *init_selector*, *fit*, *predict*, *fit_predict*, *score*.

El método *get_info* debe ofrecer información general sobre el algoritmo implementado: atributos, funcionalidad, objetivo.

El método *init_selector* es el método en el cual se realiza la inicialización del algoritmo, en caso de que sea necesario.

El método *fit* toma los datos de entrenamiento como argumentos, que pueden ser una matriz *X* en el caso del aprendizaje no supervisado, o dos matrices *X* e *Y* en el caso del aprendizaje supervisado, y se ajusta el modelo. El modelo en el método de ajuste puede ser tan simple como resolver una ecuación dada o tan complicado como un modelo de aprendizaje automático. El método devuelve el resultado a través del parámetro *self*.

El método *predict* acepta como datos de entrada los atributos *X*. A partir de los parámetros generados en el método *fit*, se realiza una predicción sobre nuevos datos presentados al algoritmo y se devuelve el resultado de la predicción.

El método *fit_predict* realiza el *fit* y *predict* en un mismo paso. Los datos de entrada son los atributos *X* y el objetivo *Y*, y devuelve el resultado de la predicción.

El método score introduce la función de evaluación, a través de *class_genetic_parameters*, que se usará como métrica cuando el algoritmo implementado se usa en un algoritmo genético.

### 4.1.3. Datos de entrada/salida de los modelos

La entrada de los modelos y transformaciones, o tuberías, siempre serán un dataframe de Dask. En cuanto a las particiones, se considera que una buena práctica es no multiplicar innecesariamente el número de particiones, particiones de unos 100MB puede ser una buena elección.

*dataframe.repartition(partition_size='100MB')*

En cuanto a las salidas. Todos los modelos deben producir arrays de Dask, incluso en el hipotético caso de que sean un único valor. Por su lado, las transformaciones deben generar un dataframe de Dask, bien heredando las columnas de la entrada o generando nuevas columnas, en el caso de que la transformación genere nuevos atributos o incluso un nuevo dataframe.

### 4.1.4. Consejos de buenas prácticas

#### 4.1.4.1. Uso de decoradores

Los decoradores son muy útiles para reutilizar código que desempeñan tareas comunes, y ayudan a que nuestro código sea más corto y limpio. En AItenea seguimos en esta línea y hemos implementado decoradores. Tanto en las clases de transformación como en las clases de *machine learning* los módulos: *fit, fit_transform y fit_predict* deben incluir *@fit_decorator*. Es una función que asegura la coherencia, entre la entrada y la salida de los datos en cada tarea del flujo de simulación.

#### 4.1.4.2. Clases auxiliares

Es aconsejable que los métodos de las clases implementadas en AItenea sean concisos, sencillos, con pocas líneas de código. Si el algoritmo que se desea implementar comprende cálculos laboriosos se recomienda construir una clase auxiliar en la que se desarrolle parte de la algoritmia necesaria. Exponemos un breve ejemplo ilustrativo:

```
from aitenea.aitenea_core.base_class_preprocessing import BaseClassPreprocessing
from aitenea.logsconf.log_conf import logging_config
from aitenea.aitenea import _MiClaseAux
```

```
class MiClase(BaseClassPreprocessing):
   options = {
       "parameter1": {"type": " ","default": " ", "gen": " "},
       "parameter2":{"type": " ","default":" ", "gen":" "},
   }
   def __init__(self, user_parameters, user_genetic_parameters=None):
       class_options = {
         "parameter1": {"type": " ","default": " ", "gen": " "},
         "parameter2":{"type": " ","default":" ", "gen":" "},
       }
       self.class_parameters = {'options': class_options}
       super(MiClase, self).__init__(
             self.__class__.__name__, self.class_parameters, user_parameters)
       self.mind = _MiClaseAux()

   def get_info(self):
       descripción= "Información sobre la clase: parámetros, funcionalidad"
       return descripcion

   def init_selector(self):
       "Inicializar el algoritmo"
       pass

   @fit_decorator
   def fit(self, X, y=None):
       options = self.parameters_values['options']
       param1 = options["parameter1"]
       param2 = options["parameter2"]
       self.mind.set_parameter(param1,param2)
       return self

   @fit_decorator
   def fit_transform(self, X, y=None):
       self.fit(X)
       return self.transform(X)

   def transform(self, X):
       return self.mind.calculate(X)


class _MiClaseAux(object):
  def __init__(self):
     "inicializar los atributos del objeto que se crea"
  def set_parameter(self,param1,param2):
     "Establecer valor de los parámetros del modelo"
     return
  def calculate(self,X):
     "Implementación de la algoritmia y devolver resultado del modelo"
     return
```

Los métodos *set_parameter* y *calculate*, de la clase auxiliar *_MiClaseAux* tienen la funcionalidad, el primero de establecer los valores por defecto de los parámetros del modelo, y el segundo, implementar la algoritmia del modelo.

### 4.1.5. Ejemplo implementación de algoritmos

En el siguiente ejemplo mostramos cómo se ha implementado el algoritmos K-Means. Siendo un algoritmo de *machine learning* hereda de la clase *base_class_ai.py* y se realizan los *import* necesarios. Contiene obligatoriamente, tal como es requerido por la clase base, los métodos *get_info*, *init_selector*, *fit*, *predict*, *fit_predict*, *score*. Como mejora se ha incluido el método *calculate_optima_nclasses* que dado un rango (1, *max_class*), *max_class* siendo el número máximo de agrupaciones, devuelve el número óptimo de *clusters*.

Los parámetros del algoritmo y los parámetros genéticos, si existieran, se deben escribir como una variable de clase.

A continuación describimos cómo se deben escribir estas variables y qué campos deben contener:

- Debe ser un diccionario: *options* = {}
- Cada variable (hiperparámetro) debe ser una entrada (*key*) del diccionario anterior que a su vez debe contener un diccionario con los siguientes campos:
    - ➢ *type*: tipo python para la variable, si se trata de una lista de opciones el tipo debe ser *list*. Los tipos aceptados son los siguientes: *int*, *float*, *bool*, *str*, etc. https://www.w3schools.com/python/python_datatypes.asp
    - ➢ *range*: [*min, step, max*] si el máximo y el mínimo coinciden no hay límites para la variable, si el paso en *None* el valor puede coger cualquier valor válido para su tipo. Si el tipo es *list* el rango es una lista con todos los valores posibles, ya sean números, *strings* o mezcla de ambos.
    - ➢ *default*: valor por defecto de híperpárametro.
    - ➢ *gen*: booleano, que indica si este híperparámetro se usará (*True*) o no (*False*) como híperparámetro genético.

La función *fitness*, que determinará la selección de los mejores individuos que seguirán evolucionando en un algoritmo genético, se debe introducir de la siguiente forma:

- Debe ser un diccionario: *genetic_parameters*
- Debe contener la entrada (*Key*) *fitness_function*, que debe contener un diccionario con los siguientes campos:
    - ➢ *type*: tipo python para la variable (*list, int, etc.*.).
    - ➢ *range*: rango de posibles valores.
    - ➢ *default*: valor por defecto.

```
from dask_ml.cluster import KMeans
from sklearn.metrics import calinski_harabasz_score

from aitenea.aitenea_core.base_class_ai import BaseClassAI
from aitenea.logsconf.log_conf import logging_config
from aitenea.exceptions.exceptions import NotFitError
from aitenea.aitenea_core.decorators import fit_decorator
```

```
import logging
from logging.config import dictConfig

loggtype = 'CONSOLE'

dictConfig(logging_config)
logger = logging.getLogger(loggtype)

class Kmeans(BaseClassAI):
    options = {
            'num_cluster':
            {"type": "int", "range": [1, 1, 1],
             "default": 2, "gen": True},
            'method':
            {"type": "list", "range": ["k-means||", "k-means++"],
             "default": "k-means++", "gen": True},
            "auto_optimal_cluster":
            {"type": "bool", "range": None, "default": False, "gen": False}, }

    genetic_parameters = {"fitness_functions": {"type": "list", "range": [
            "calinski-harabasz"], "default": ["calinski-harabasz"]}}

    def __init__(self, user_parameters, user_genetic_parameters=None):
        """
        Algorithm for classification, K-Means

        Arguments:
            BaseClassAI {[type]} -- [description]
            user_parameters {[type]} -- [description]
        """
        class_options = {
            'num_cluster':
            {"type": "int", "range": [1, None, 1],
             "default": 2, "gen": True},
            'method':
            {"type": "list", "range": ["k-means||", "k-means++"],
             "default": "k-means--", "gen": True},
            "auto_optimal_cluster":
            {"type": "bool", "range": None, "default": False, "gen": False}, }
        self.class_parameters = {'options': class_options}
        class_genetic_parameters = {"fitness_functions": {"type": "list", "range": [
            "calinski-harabasz"], "default": ["calinski-harabasz"]}}
        self.class_genetic_parameters = {"options": class_genetic_parameters}
        super(
            Kmeans, self).__init__(
            self.__class__.__name__, self.class_parameters, user_parameters,
            self.class_genetic_parameters, user_genetic_parameters)
        self.mind = None

    def get_info(self):
        """
        Gives information about how to use this class and its parameters
        """
        description = """ """
        return description

    def init_selector(self):
        """
        Method used for initialization, if necessary
        """
        pass
```

```
    def transform(self, X):
        """
        The parameters generated from the
        *fit* method, applied on the model return a set with
        transformed data.

        Arguments:
            X (dask): features matrix
        """
        pass

    @fit_decorator
    def fit(self, X, y=None):
        """
        Is responsible for estimating the attributes of the input
        data and storing the attributes of the model and finally
        returning the result

        Arguments:
            X (): features matrix

            y (dask): target variable (default: {None})
        """
        logging.info("Init clustering fit")
        options = self.parameters_values['options']
        automatic = options['auto_optimal_cluster']
        method = options['method']
        num_cluster = options['num_cluster']
        if automatic:
            if num_cluster <= 2:
                num_cluster = 2
            else:
                num_cluster = self.calculate_optima_nclasses(
                    X, method, num_cluster)
        logger.info("Start K-Means clustering for %s class", num_cluster)
        self.mind = KMeans(n_clusters=num_cluster,
                           init=method, random_state=0)
        self.mind.fit(X)
        return self

    def fit_transform(self, X, y=None):
        """
        Performs *fit* and *transform* in one step

        Arguments:
            X (dask): features matrix

            y (dask): target variable (default: {None})

        Returns:
            (dask): transformed dataset
        """
        logging.info("Init clustering fit_transform, fit without effect")
        self.fit(X)
        return self.mind.predict(X)


    def fit_predict(self, X, y=None):
        """
        Performs *fit* and *predict* in one step

        Arguments:
```

```
            X (dask): features matrix

            y (dask): target variable (default: {None})

        Returns:
            (dask): dataset with the result of the prediction
        """
        logging.info("Init clustering fit_transform, fit without effect")
        return self.fit_transform(X)

    @fit_decorator
    def predict(self, X):
        """
        The model generated in *fit* method is used to make prediction
        on new data

        Args:
            X (dask): features matrix

        Raises:
            *NotFitError*: launches an error if model was not
            previously fitted.

        Returns:
            (dask): dataset with the resulting prediction
        """
        logging.info("Init clustering predict")
        try:
            return self.mind.predict(X)
        except Exception as err:
            logger.error("Error to fit transform, %s", err)
            raise NotFitError

    def score(self, X, labels):
        """
        The evaluation function which will be used as a metric
        when the implemented algorithm is used in a genetic algorithm

        Args:
            X (dask): dataset

            labels (array): predicted labels for each sample.

        Returns:
            (float): the resulting evaluation score
        """
        genetic_options = self.genetic_parameters_values['options']
        fitness_fun = genetic_options["fitness_functions"]
        if fitness_fun == "calinski-harabasz":
            return {
                "calinski_harabasz_score": calinski_harabasz_score(X, labels)}

    def calculate_optima_nclasses(
            self, dataframe, method, max_class, tolerance=0.2):
        """
        Method to automatically calculate the number of optimal
        clusters *k*

        Arguments:
            dataframe (dask): dataset

            method (str): method for initialization
```

```
            max_class (int): maximum number of clusters

        Keyword Arguments:
            tolerance (float): tolerance level (default: {0.2})

        Returns:
            (int): number of optimum clusters
        """
        wcss = []
        slope = []
        for i in range(1, max_class):
            brain = KMeans(n_clusters=i, init=method,
                           random_state=0)
            brain.fit(dataframe)
            wcss.append(brain.inertia_)
        for n in range(0, len(wcss)-1):
            slope.append(wcss[n]-wcss[n+1])
        min_slope = min(slope)
        position_min = slope.index(min_slope)
        for n in range(position_min-1, -1, -1):
            if abs(slope[n] - min_slope) < min_slope*tolerance:
                return n
        return position_min
```

## 4.2. Implementación de nuevos nodos

Tal como hemos detallado en 3.3. Capa de alto nivel, hemos implementado nuevos nodos para cubrir con la funcionalidad deseada para nuestro *framework.* Estos nodos tienen un propósito muy bien definido, y se han agrupado según la funcionalidad que desempeñan para una mejor visibilidad y claridad.

Como ejemplo de codificación de uno de estos nodos y siguiendo la arquitectura descrita anteriormente para la capa de alto nivel *aitenea_node-red*, presentamos en este caso el nodo *Manual-data*, mostrando ejemplo de los ficheros que los define:

**manual-data.js**

```
module.exports = function (RED) {
  function ManualData(config) {
    RED.nodes.createNode(this, config);
    let node = this;
    this.data = config.data;

    node.on('input', function () {
      let data = {
        "origin": {"type": "manual data", "options": JSON.parse(this.data)},
      };
      let msg =  {"payload": data};
      node.send(msg);
```

```
    });

  }
  RED.nodes.registerType("Manual-data", ManualData);

  RED.httpAdmin.post("/inject/:id", RED.auth.needsPermission("inject.write"), function (req, res) {
    let node_origin = RED.nodes.getNode(req.params.id);
  });
}
```

### manual-data.html

```
<script type="text/javascript">
  RED.nodes.registerType("Manual-data", {
    category: "AItenea Connector",
    color: "#14E81B",
    defaults: {
      data: {
        value: "",
        required: true,
      },
    },
    inputs: 0,
    outputs: 1,
    icon: "batch.svg",
    label: function () {
      return "Manual-data" || this.name;
    },
    outputLabels: function () {
      return "To Model-get or Model-set " + this.data;
    },
    oneditprepare: function (msg) {
      $("#node-input-data").typedInput({
        type:"json",
        types:["json"]
      });
    },
    button: {
      enabled: function () {
        return true;
      },
      onclick: function () {
        var node = this;
        $.ajax({
          url: "inject/" + this.id,
          type: "POST",
          contentType: "application/json; charset=utf-8",
          success: function (resp) {
            console.log("SUCESSS", resp);
          }
        });
      },

    }
  });
</script>

<script type="text/html" data-template-name="Manual-data">

<div class="form-row node-text-editor-row">
```

```
  <label for="node-input-data">
    <i class="fa fa-pencil" aria-hidden="true"></i>Data in JSON</label>
  <input type="text" id="node-input-data"  placeholder="{'X':{'A':[1]}}" >
</div>
</script>

<script type="text/html" data-help-name="Manual-data">
  <p>
    This node enters numeric values by hand in json representation
  </p>
</script>
```

### package.json

```
{
    "name": "manual-data",
    "version": "1.0.0",
    "description": "",
    "main": "index.js",
    "scripts": {
      "test": "echo \"Error: no test specified\" && exit 1"
    },
    "node-red" : {
      "nodes": {
          "Manual-data": "manual-data.js"
      }
    },
    "author": "Jose Luis Blanco",
    "license": "ISC",
    "dependencies": {
      "request": "2.88.2"
    }
  }
```