Cómo Extender DashAI

I. Introducción

A modo de preparación para extender el software, te presentamos DashAI y te explicamos con más detalle su funcionamiento.

Qué es DashAI

DashAI es un software de código abierto que permite entrenar modelos de aprendizaje automático (ML) mediante una interfaz gráfica interactiva, cubriendo desde la carga de datos hasta la evaluación de resultados.

El software está diseñado para poder integrar y ser compatible con diversas bibliotecas científicas de ML populares, aprovechando así los recursos existentes en el ecosistema de IA. Dicho de otra forma, DashAI apunta a agrupar distintas herramientas como scikit-learn, pytorch, tensorflow o huggingface en un solo software y que la gente pueda utilizarlas para realizar experimentos y entrenamientos de manera sencilla a través de su interfaz gráfica.

Arquitectura

El sistema de extensiones de DashAI se basa en una arquitectura de plugins. Esto consta de, por un lado, un ***core*** que contiene todo lo necesario para orquestar las etapas del flujo de ML y, por otro lado, existen módulos instalables llamados ***plugins*** que empaquetan distintos componentes previamente definidos que se pueden extender: modelos, tareas, dataloaders y métricas de evaluación.

Para que veas cómo funciona DashAI y la instalación de plugins, ve el siguiente video: https://youtu.be/4G0CrX2D8yg

Se espera que personas con un conocimiento avanzado en ML puedan crear *plugins* que extiendan el software y permitan que usuarios finales puedan entrenar distintos modelos de ML, realizando experimentos, configurando sus parámetros y evaluando su desempeño. Para que el proceso de crear una extensión sea sencillo, se diseñaron los mecanismos necesarios para que los plugins se integren fácilmente al sistema y que se puedan desplegar correctamente en la interfaz gráfica: incluso para habilitar la configuración de hiperparámetros desde la interfaz gráfica solamente se debe declarar en el plugin mismo, sin necesidad de hacer modificaciones en el frontend del software.

Componentes y el pipeline

Un componente es una **clase de python que pertenece a uno de los posibles tipos: modelo, tarea, dataloader o métrica de evaluación,** y que cumplen con implementar ciertos métodos (y sus respectivas firmas) y declarar atributos propios de cada uno de los tipos de componente. Los distintos componentes son usados por el software para articular y

permitir al usuario completar el flujo de ML desde la carga de datos hasta la evaluación de resultados.

El pipeline de ML supervisado que sigue el software es el siguiente: primero, se carga un dataset (e.g. archivo CSV) mediante un componente de tipo **dataloader** y se guarda como un Dataset de la librería Huggingface[1], que es el formato de dataset estándar que DashAI utiliza de manera interna. Luego, el usuario configura su experimento y realiza el entrenamiento (llamada al método *fit)* de una o más instancias de las clases de tipo **model** con los parámetros ingresados por el usuario. Posteriomente, sus predicciones (llamadas al método *predict*) son evaluadas usando las métricas disponibles para la tarea, definidas en los componentes de tipo **metric**. Tanto el método *fit* como *predict* reciben su data respectiva en el formato dataset de Huggingface[2] guardado anteriormente.

Tanto componentes de tipo **model** y **metric,** necesitan señalar explícitamente los nombres de la o las *task* a las que están asociados dichos componentes[3]. De esta manera, DashAI sabe qué componentes están disponibles para la tarea que el usuario seleccionó.

Por último, aquellos componentes que son configurables por el usuario mediante la interfaz gráfica son denominados “Objetos Configurables” y deben heredar de la clase ConfigObject, así como también deben tener asociado un esquema de *pydantic* que declare el tipo de dato que espera y permita su despliegue y visualización en la interfaz gráfica como un campo que recibe un input por parte del usuario.

Plugins

Un plugin de DashAI corresponde a un **paquete de python que contiene uno o más componentes extensibles** y que es reconocible por el software mediante el uso de *entrypoints*[4]. Esto implica que el empaquetamiento de los componentes en un plugin es totalmente libre: puede contener todos los componentes necesarios para una tarea en particular o un plugin puede ser simplemente un modelo específico o un dataloader para algún tipo de dataset determinado.

Los plugins están pensados para ser compartidos mediante el repositorio de paquetes de python (*pypi*) y a la vez, seleccionables e instalables desde la propia interfaz gráfica de DashAI.

---

[1] https://huggingface.co/docs/datasets/index
[2] Notar que parte de lo que el modelo debe implementar, es adaptar este tipo de datos a un formato compatible con la librería que el modelo esté utilizando.
[3] Para ello deben contener el atributo “COMPATIBLE_COMPONENTS”.
[4] Un entrypoint es un tipo de metadata que puede ser expuesta por los paquetes de python en su instalación. Resultan útiles si un programa busca la personalización a través de *plugins*.

II. Construir un plugin

En esta sección se explicará paso a paso cómo construir un plugin. En su estructura, este es un paquete de python, por lo que debe ser definido como tal. Esto implica también que es instalable en el computador del usuario mediante el instalador de paquetes de python (*pip*).

Si bien un plugin es un paquete de python, no cualquier paquete de python es un plugin. Para que así sea, debe cumplir ciertas reglas que DashAI define. Por un lado, **un plugin debe contener uno o más *componentes***, es decir, clases que definan un modelo, tarea, dataloader o métrica. Dicha clase, a su vez, debe cumplir con una interfaz determinada (métodos y sus firmas) dependiendo del tipo de componente que sea. Si tiene parámetros que el usuario puede modificar, entonces debe tener asociado un esquema. Por otro lado, en la configuración del paquete se debe declarar al componente como un *entrypoint*[5] para que el software lo pueda identificar, registrar e integrar para su posterior uso.

a) Estructura de un paquete de python

⚡ Puedes encontrar un template con la estructura en "*dashai_plugin_template*". Es importante que el nombre del plugin comience con el prefijo "*dashai_*".

La estructura de directorios que debe tener el paquete se presenta a continuación:

```
nombre_del_plugin
|   LICENSE
|   pyproject.toml
|   readme.md
+---src
    +--- nombre_del_plugin
          |  example_model.py
```

A continuación se desglosan sus elementos:

1. **src:** Contiene la carpeta que se copia en el directorio de paquetes de python. Esta última debe tener el nombre del paquete o plugin y contener todos los archivos necesarios para extender el software. **Es importante que el nombre del paquete comience con el prefijo "*dashai_*"** para su posterior identificación.

2. **pyproject.toml:** Archivo de configuración. Este es el encargado de determinar la forma en que se crea el paquete y sus metadatos.

[5] Un entrypoint es un tipo de metadata que puede ser expuesta por los paquetes de python en su instalación. Resultan útiles si un programa busca la personalización a través de *plugins*.

3. **readme.md:** Contiene la descripción larga del paquete o plugin. Se muestra en pypi y en la vista en la interfaz del plugin.

4. **LICENSE (opcional):** Especifica el tipo de licencia bajo la que se distribuirá el paquete.

A continuación se detallará cómo crear las clases de un componente que deben ir en el directorio del paquete (dentro de la carpeta *src)*, así como la estructura del *pyproject.toml*.

b) Clase del componente

⚡ Puedes encontrar templates en la carpeta *"src/dashai_plugin_template"*. En ellos se especifican los métodos (y sus firmas) que deben implementar según el tipo de componente que se quiere extender.

Un componente es una clase de python que pertenece a uno de los posibles tipos: modelo, tarea, dataloader o métrica de evaluación. Para declarar que una clase es un componente, debe heredar de una de las clases Base: *BaseModel*, *BaseTask*, *BaseDataloader* y *BaseMetric*[6]. Estas clases se pueden encontrar en el core de DashAI. La implementación de **los componentes debe contener los métodos y atributos definidos para cada tipo de componente**. Esto permite su integración con el software.

Los componentes (que no son tasks) son *Task-oriented*, es decir, pueden ser usados en el contexto de una o más tareas. Estos deben contener el atributo "COMPATIBLE_COMPONENTS" que corresponde a una lista con los nombres (*str*) de las clases de la(s) tarea(s) asociadas al componente.

Por otro lado, los componentes que pueden ser configurables a través de la interfaz gráfica de usuario, deben heredar de *ConfigObject*[7] y deben contener el atributo "SCHEMA" señalando la clase del esquema asociado al componente. Por ejemplo, en el caso de un componente que implementa un modelo KNN de clasificación tabular:

```python
class KNeighborsClassifier(BaseModel, ConfigObject):
    """Scikit-learn's K-Nearest Neighbors (KNN) classifier wrapper for DashAI."""

    SCHEMA = KNeighborsClassifierSchema
    COMPATIBLE_COMPONENTS = ["TabularClassificationTask"]

    def __init__(self, **kwargs) -> None:
```

[6] Estas clases base se encuentran y deben ser importadas desde el core de DashAI.
[7] Idem anterior. Se encuentra en DashAI.back.config_object.ConfigObject

c) Esquemas de pydantic

⚡ Los componentes configurables mediante la interfaz gráfica deben definir una clase que herede de BaseSchema (pydantic). En ella, se debe especificar el tipo de campo y sus restricciones para cada parámetro que será desplegado en la GUI.

DashAI usa *pydantic*[8] para definir y validar el tipo de input que es posible ingresar en cada parámetros configurable. Así, se despliegan advertencias al usuario en la interfaz gráfica cuando está ingresando un valor inválido.

Para que este mecanismo opere correctamente, es necesario definir el tipo y las restricciones de cada parámetro en una clase que herede de *BaseSchema* (clase incluída en el *core* de DashAI) y que se utilicen los objetos *fields* para señalar el tipo de parámetro que corresponda. Estos se encuentran en el *core* de DashAI y algunos ejemplos son: *int_field*, *bool_field*, *enum_field*, *float_field*, *list_field*, *string_field*. Para un listado completo y documentación, ver Anexo 1.

A modo de ejemplo, parte del esquema para un modelo K-Nearest-Neighbor (KNN), luce así:

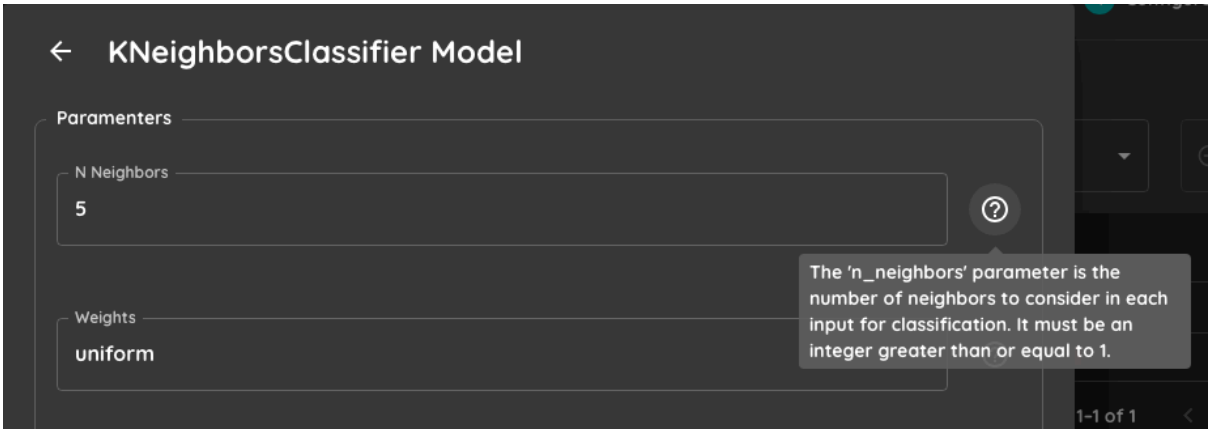
```python
class KNeighborsClassifierSchema(BaseSchema):
    n_neighbors: schema_field(
        int_field(ge=1),
        placeholder=5,
        description="The 'n_neighbors' parameter is the number of neighbors to "
        "consider in each input for classification. It must be an integer greater "
        "than or equal to 1.",
    ) # type: ignore

    weights: schema_field(
        enum_field(enum=["uniform", "distance"]),
        placeholder="uniform",
        description="The 'weights' parameter must be 'uniform' or 'distance'.",
    )  # type: ignore
```

Se muestran dos parámetros del esquema. El primero, el número de vecinos, corresponde a un field de tipo *int*, que debe ser mayor que 1. Además, el placeholder, o valor por defecto, es 5. El texto de la descripción se mostrará en la interfaz. El segundo es un field del tipo *enum*, es decir, una lista desplegable con valores determinados, en este caso, puede ser “uniform” o “distance”. Estos parámetros definidos en el esquema lucen así en la interfaz:

---

[8] Pydantic es una biblioteca de validación de datos y gestión de tipos para Python, que utiliza anotaciones de tipo para definir de manera declarativa y validar estructuras de datos, proporcionando una validación automática y robusta.



4) Archivo de configuración (pyproject.toml)

⚡ En el archivo pyproject.toml se debe declarar el entrypoint del componente bajo el grupo 'dashai.plugins'. Además, se deben incluir los tipos de componentes en el campo keywords.

Este archivo contiene la descripción general del paquete, como el nombre, la versión y la información del autor. A continuación se presentarán aspectos importantes a considerar del archivo de configuración:

- Declarar entry point

```
[project.entry-points.'dashai.plugins']
ExampleClass = 'plugin_name.example_class:ExampleClass'
```

Cada una de las clases que implementan un componente y que estén incluídas en el plugin, se deben declarar como *entrypoints* en el grupo 'dashai.plugins'. De esta manera, DashAI puede saber que están disponibles e incluirlas dentro del flujo de ML. La estructura del entrypoint es la que se señala arriba. Para más información, visitar este link.

- Definir keywords

En la sección '[project]' del archivo de configuración se debe incluir una variable keywords que corresponde a una lista con los tipos de componentes que contiene el plugin.

- Versión del paquete

En la sección '[project]' también se incluye la versión del paquete. Es importante que cada vez que se realicen cambios en el plugin se actualice su versión. De esta manera, se podrán subir dichos cambios a pypi correctamente.

II. Subir un plugin a pypi

Flujo para cargar en pypi

El siguiente tutorial usa twine para subir el paquete a pypi. Sin embargo, hay muchas otras formas de hacerlo, y se tiene la libertad de usar cualquiera.

- Empaquetar archivos

```
python -m pip install --upgrade build
python -m build
```

Esto creará una distribución como la siguiente:

```
+---dist
      |  nombre-plugin-0.0.1--none-any.whl
      |  nombre-plugin-0.0.1.tar.gz
```

- Token de pypi

Con una cuenta creada en pypi.org, se puede solicitar un token que permite subir los paquetes al repositorio. Para ello, en la configuración de su cuenta, vaya a la sección Fichas de API y seleccione «Añadir ficha de API». Para más información, visitar esta página.

Para usar twine, se debe crear un archivo "*.pypirc*" en $HOME que incluya el token anteriormente generado. Instrucciones más precisas se pueden encontrar acá.

- Subir paquete a pypi

```
python -m pip install --upgrade twine
python -m twine upload --repository pypi dist/*
```

Glosario

Componente: Objeto individual que extiende DashAI. En concreto, es una clase de python que puede ser registrada en el software y utilizada sin errores en alguna parte del flujo de ML. Si es configurable desde la UI debe también tener asociado un esquema.

Tipo de componente: Los componentes pueden pertenecer a la categoría de métrica, modelo, dataloader o task.

Plugin: Paquete de python que contiene uno o más componentes extensibles.

Objeto configurable: Corresponde a un componente que contiene parámetros que el usuario puede seleccionar y configurar mediante la interfaz gráfica. Para esto, es necesario que se declare un esquema que defina el tipo de parámetro que se espera recibir.

Anexo 1: Tipos de fields

> DashAI.back.core.schema_fields.bool_field()

Para declarar un parámetro booleano

> DashAI.back.core.schema_fields.component_field(*parent)*

Para declarar un parámetro que espera un componente

- *parent : str*

  The name of the component's parent class is used to select the components to show the user.

> DashAI.back.core.schema_fields.enum_field(*enum*)

Para declarar un parámetro que espera un string de una conjunto de valores

- *enum*: *List[str]*

  All the posible string values of the field.

> DashAI.back.core.schema_fields.float_field(*ge, gt, le, lt)*

Para declara un parámetro numérico de tipo floating-point

- *ge: Optional[float], default = None*

  An optional float that the value should be greater than or equal to. If not provided, there is no lower limit.

- *gt: Optional[float], default = None*

  An optional float that the value should be strictly greater than. If not provided, there is no strict lower limit.

- *le: Optional[float], default = None*

  An optional float that the value should be less than or equal to. If not provided, there is no upper limit.

- *lt: Optional[float], default = None*

  An optional float that the value should be strictly less than. If not provided, there is no strict upper limit.

> DashAI.back.core.schema_fields.int_field(*ge, gt, le, lt)*

Para declarar un entero

- *ge: Optional[float], default = None*

  An optional float that the value should be greater than or equal to. If not provided, there is no lower limit.

- *gt: Optional[float], default = None*

  An optional float that the value should be strictly greater than. If not provided, there is no strict lower limit.

- *le: Optional[float], default = None*

  An optional float that the value should be less than or equal to. If not provided, there is no upper limit.

- *lt: Optional[float], default = None*

  An optional float that the value should be strictly less than. If not provided, there is no strict upper limit.

> DashAI.back.core.schema_fields.none_field(*t*)

Para declarar un parámetro opcional, es decir, su valor puede ser *none*

- *t: Type*

  The type of field that is going to be optional

> DashAI.back.core.schema_fields.string_field()

Para declarar un parámetro de tipo string

> DashAI.back.core.schema_fields.union_type(*t1, t2*)

Para declarar un parámetro que puede tener dos tipos diferentes

- *t1: Type*

  The first field's type

- *t2: Type*

  The second field's type

> DashAI.back.core.schema_fields.list_field(*item_type, min_items, max_items*)

Para declarar un parámetro que es una lista de elementos

- *item_type: Type*

  The list elements field's type (e.g. string_field() in case of a string list)

- *min_items: Optional[int], default = None*

  An optional integer that defines the minimum number of items the list must have

- *max_items: Optional[int], default = None*

  An optional integer that defines the maximum number of items the list must have