PEC 2

Álvaro Maestre Santa

Explicación Ejercicios:

1. **Crear el pipeline de ML (con transformers y estimators) para dar un resultado (los labels están definidos arriba)**

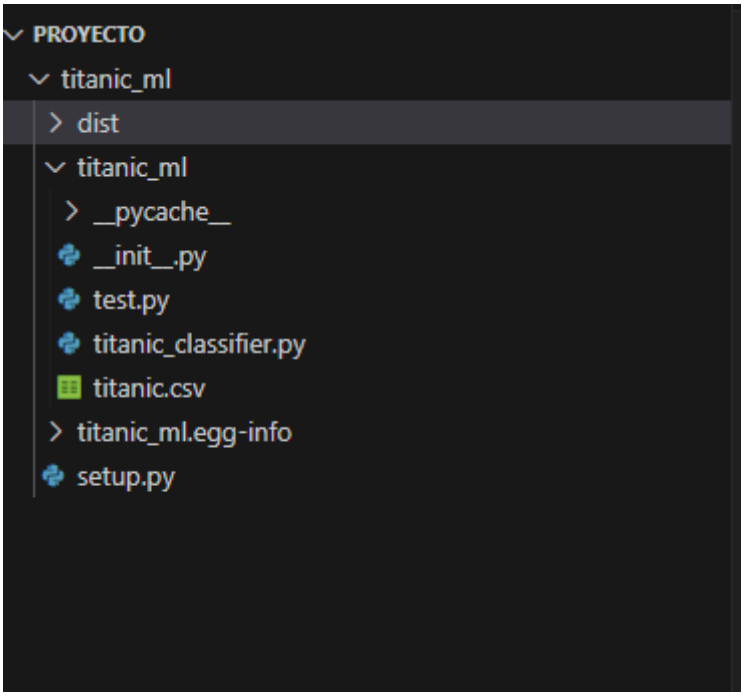
   El código está en el archivo ***Ejercicio1_código.py.***

2. **Empaquetar el algoritmo, dos formas posibles a elección:**

   a. Como un archivo de tipo pkl
   b. Como una librería disponible en PiPy

En este caso me he decantado por hacerlo mediante la segunda opción, para ello lo que he hecho ha sido lo siguiente:

1. Crear una estructura de carpetas, en mi caso fue la siguiente:



2. Dentro de titanic_classifier, metí el algoritmo usado para hacer el ejercicio 1, aunque en este caso, al ser .py, lo he retocado haciendo una clase TitanicClassifier, que tiene dos métodos, fit y predict. El método fit realiza todo el preprocesamiento de los datos, entre el modelo de clasificación y construye el pipeline. El método predict utiliza el pipeline entrenado para hacer predicciones.
3. Dentro del archivo setup.py, rellené con los datos necesarios para elaborar el empaquetamiento.

4. Creé el paquete usando el siguiente comando: python setup.py sdist.
5. Finalmente, subo el paquete con el comando: twine upload dist/*. Al ejecutar este comando, me pide el usuario y la contraseña de pypi.org, por lo que me creé una cuenta en dicha página. Dicha librería se puede ver en el siguiente enlace: Librería pypi

Finalmente, para poder usar la librería bastaría con hacer un ***pip install titanic_ml***

3. Disponibilizar el algoritmo como API utilizando Flask dentro de un Contenedor y subirlo a Docker Hub.

En este caso, para resolver el ejercicio he seguido los siguientes pasos:

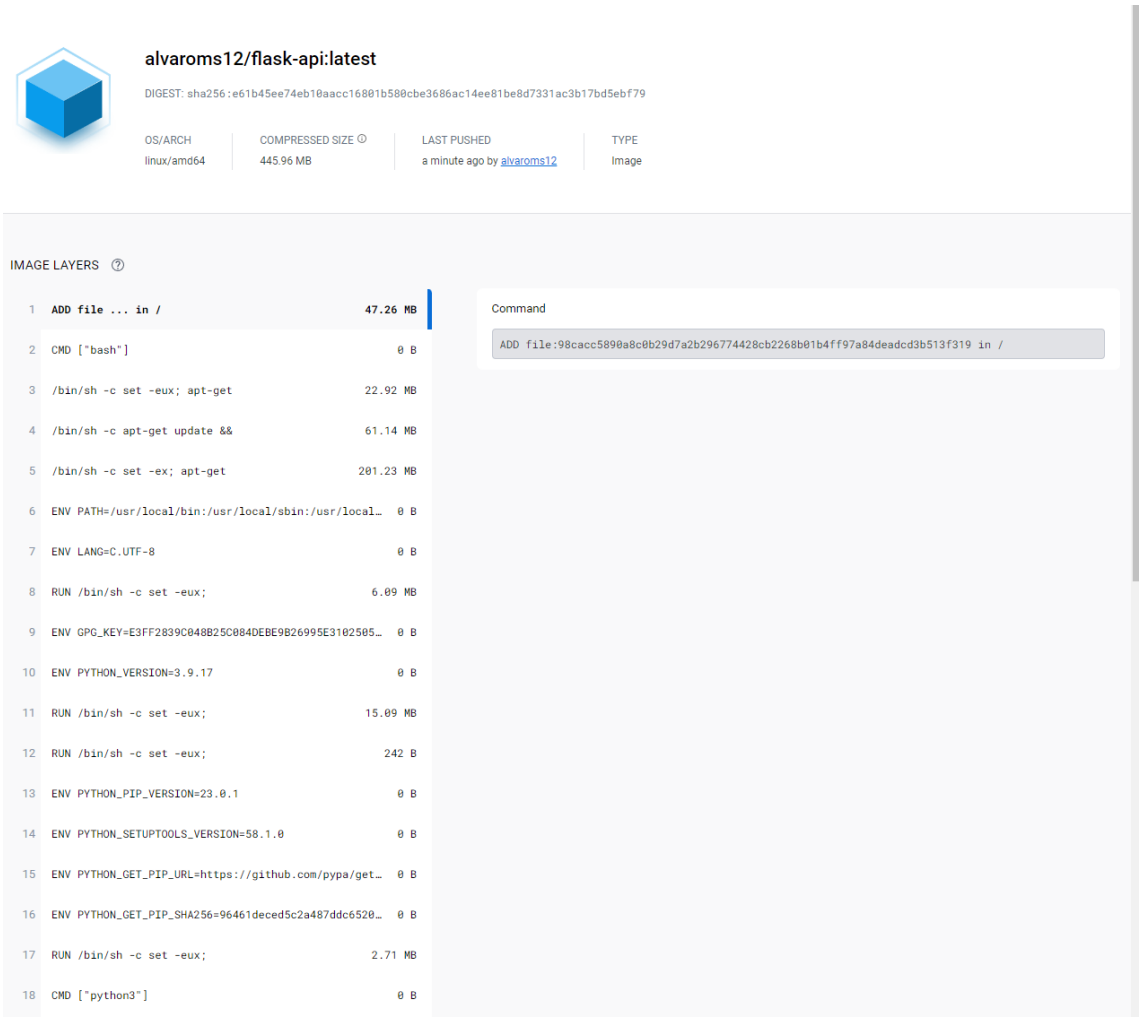
1. He creado un archivo app.py donde contendrá el algoritmo que he usado para resolver el ejercicio 1.
2. Para convertir el fichero app.py en una API con Flask, lo primero que he hecho ha sido importa las librerías de flask. Posteriormente he creado una instancia de la aplicación Flask, además de una función llamada predict, que contiene el algoritmo usado en el ejercicio 1, pero en vez de imprimir el resultado, nos devuelve el resultado en un JSON para verlo en la Web. Por último, al final de la aplicación pongo unas líneas que harán que se ejecute la aplicación Flask.
3. Para crear el contenedor, he generado un archivo Dockerfile, donde se especifica cómo se va a construir el contenedor.
4. Por último, he generado un archivo txt llamado, requirements para indicar las librerías que se usan en mi algoritmo.
5. Por último, sólo basta construir el contenedor, con los siguientes comandos:

```
docker build -t nombre_usuario/nombre_imagen:etiqueta .
docker run -d -p 5000:5000 --name nombre_contenedor
nombre_usuario/nombre_imagen:etiqueta
```

6. Como hay que subirlo a docker hub, sólo hay que ejecutar el siguiente comando:

```
docker push nombre_usuario/nombre_imagen:etiqueta
```

Con estó he podido subir el contenedor a Docker Hub, como se puede ver en la siguiente imagen:



Además, adjunto el link del contenedor: Link Contenedor

El comando para hacer un pull es el siguiente:

docker pull nombre_usuario/flask-api

Explicación del algoritmo usado en la API Flask:

1. Importación de módulos:
    - Importo los módulos necesarios para construir la aplicación Flask y realizar la tarea de clasificación.
    - **Flask** se utiliza para crear la aplicación web.
    - **jsonify** se utiliza para devolver los resultados en formato JSON.
    - **pandas** se utiliza para manipular y analizar datos tabulares.
    - **Pipeline** y **ColumnTransformer** son clases de Scikit-learn utilizadas para construir el flujo de trabajo de procesamiento de datos y modelado.

- **StandardScaler**, **OneHotEncoder**, **RandomForestClassifier** y **SimpleImputer** son clases de Scikit-learn utilizadas para transformar y modelar los datos.

2. Creación de la aplicación Flask:
    - Se crea una instancia de la aplicación Flask utilizando el nombre del módulo actual como argumento.
3. Configurar la ruta web:
    - Se utiliza la línea **@app.route('/')** para asociar la función **predict()** a la ruta raíz '/' de la aplicación Flask. Esto significa que cuando se accede a la ruta raíz, se ejecutará la función **predict().**
4. Función **predict()**:
    - Dentro de la función **predict()**, se realiza la tarea de clasificación.
    - Leo los datos del archivo 'titanic.csv' en un DataFrame de pandas.
    - Utilizo **SimpleImputer** para imputar los valores faltantes en la columna 'age' con la media.
    - Divido los datos en características (**X**) y variable objetivo (**y**).
    - Divido los datos en conjuntos de entrenamiento y prueba utilizando **train_test_split**.
    - Defino las columnas numéricas y categóricas.
    - Creo transformadores (**StandardScaler** y **OneHotEncoder**) para escalar y codificar las características.
    - Combino los transformadores en un **ColumnTransformer**.
    - Creo un clasificador (**RandomForestClassifier**).
    - Creo un pipeline (**Pipeline**) que encadena el preprocesamiento y el clasificador.
    - Entreno el pipeline con los datos de entrenamiento.
    - Evalúo el rendimiento del modelo en el conjunto de prueba utilizando **score**().
    - Devuelvo el resultado de precisión (**accuracy**) en formato JSON utilizando **jsonify**().
5. Condición para ejecutar la aplicación:
    - El bloque **if __name__ == '__main__**': asegura que la aplicación Flask solo se ejecute cuando se ejecute directamente el script principal, no cuando se importe como un módulo.
6. Ejecución de la aplicación:
    - El método **run()** se utiliza para ejecutar la aplicación Flask en el puerto 8080.

4. Crear un entorno tox de pruebas dentro del repositorio que se empaquetó.

Para este ejercicio he seguido los siguientes pasos.

- Instalar **tox** en Python.
- He creado un archivo llamado **tox.ini** en el directorio raíz del repositorio, es decir, en el mismo nivel que **setup.py** y **titanic_ml**. Este archivo contiene la configuración **tox** para las pruebas, en mi caso contiene la siguiente información.

tox.ini

```
[tox]
envlist = py1, py2, py3

[testenv]
deps =
    pytest
commands =
    pytest tests

```

- En el archivo **tox.ini** he especificado tres entornos de prueba (**py1**, **py2** y **py3**) y utilizo **pytest** como dependencia para ejecutar las pruebas. Además, el comando **pytest tests** se ejecutará en cada entorno de prueba.
- Ahora ya solo queda ejecutar tox en el terminal, usando el comando: ***tox***, esto creará los entornos virtuales y ejecutará las pruebas que hemos establecido en el tox.ini, en mi caso el resultado ha sido el siguiente:

```
tests\test_titanic_classifier.py .                                       [100%]

==================================== 1 passed in 18.13s ====================================
.pkg: _exit> python C:\Users\alvar\AppData\Local\Programs\Python\Python311\Lib\site-packages\pyproje
ct_api\_backend.py True setuptools.build_meta
  py1: OK (48.84=setup[13.00]+cmd[35.84] seconds)
  py2: SKIP (0.16 seconds)
  py3: OK (161.24=setup[141.59]+cmd[19.64] seconds)
  congratulations :) (210.42 seconds)
PS H:\Mi unidad\Master\9. Analítica Escalable\Bloque 2\PEC2\proyecto\titanic_ml>
```