6. Implementación de la API con Flask

La arquitectura de la API presentada es una arquitectura de servidor basada en Flask, que es un microframework para aplicaciones web en Python. A continuación, se describen los componentes y características clave de esta arquitectura:

* Framework Web:

Flask: Se utiliza Flask para manejar las solicitudes HTTP y definir las rutas de la API. Flask es conocido por ser ligero y fácil de usar, lo que lo hace ideal para crear APIs rápidas y prototipos.

* Modelo de Machine Learning:

Joblib: Se utiliza la biblioteca joblib para cargar un modelo de Machine Learning previamente entrenado y guardado en un archivo (modelo\_entrenado.pkl). Joblib es eficiente para la serialización y deserialización de objetos de Python que contienen grandes arrays NumPy.

* Rutas de la API:

/: Ruta de prueba que devuelve un mensaje simple para confirmar que la API está funcionando.

/predict: Ruta POST que recibe datos en formato JSON, los procesa y utiliza el modelo de Machine Learning para hacer una predicción. La predicción se devuelve en formato JSON.

* Estructura de la API:

Inicio del Servidor: La aplicación Flask se ejecuta en modo de depuración (debug mode) cuando se ejecuta el script directamente.

Manejo de Solicitudes: La API maneja solicitudes POST en la ruta /predict. Los datos de entrada se esperan en formato JSON y se convierten a un array NumPy para ser utilizados por el modelo de Machine Learning.

* Manejo de Errores:

La función de predicción incluye un bloque try-except para capturar y manejar posibles errores, devolviendo un mensaje de error en formato JSON con el código de estado HTTP 400 si ocurre algún problema.

7. Integración con el Frontend

Para realizar la integración del backend basado en Flask con un frontend, puedes utilizar tecnologías como HTML, CSS, JavaScript y bibliotecas o frameworks de JavaScript como React, Vue, o Angular.

* **Pasos para la Integración del Backend con el Frontend**
  + Crear una interfaz HTML simple:

Crear un archivo index.html que contenga un formulario para ingresar los datos necesarios para la predicción.

* + Estilos con CSS:

Agregar un archivo styles.css para estilizar la interfaz.

* + Lógica del frontend con JavaScript:

Utilizar JavaScript para capturar los datos del formulario y enviar una solicitud POST a la API Flask.

Manejar la respuesta de la API y mostrar el resultado en la interfaz.

* + Utilizar un framework de JavaScript:

React, Vue o Angular pueden proporcionar una mejor estructura y funcionalidades adicionales para una interfaz más compleja.

* + Crear el Frontend en Python:

Utilizar una biblioteca como requests para enviar datos a la API Flask y recibir la respuesta.

* Proceso de Consumo de la API:

Para consumir la API, se utilizó Python. Los pasos son los siguientes:

* + Utilizar la biblioteca requests para enviar una solicitud POST a la API Flask.
  + Enviar los datos necesarios para la predicción en el cuerpo de la solicitud en formato JSON.
  + Manejar la respuesta de la API imprimiendo la predicción en la consola o mostrando un mensaje de error en caso de que ocurra algún problema.

8. Pruebas y Validación

Para la desarrollo de pruebas se realizó un mapeo de valores de la columna “Tipo de Discapacidad”, obteniendo los siguientes resultados:

0: 'INTELECTUAL'

1: 'AUDITIVA '

2: 'FISICA'

3: 'INTELECTUAL '

4: 'Sin Discapacidad'

5: 'PSICOSOCIAL'

6: 'VISUAL'

7: 'AUTISMO'

8: 'MIXTA'

* Pruebas del Modelo:
  + Resultados de las pruebas del modelo.

Del total de datos del data set se dividió de la siguiente manera:

* Para el conjunto de entrenamiento se tomaron un total de 115 datos
* Para el conjunto de prueba se tomaron 50 datos

Con los cuales se obtuvieron los siguientes resultados:

Accuracy: 1.0

Precision: 1.0

Recall: 1.0

F1 Score: 1.0

* Con una exactitud de 1.0, significa que el modelo predijo correctamente el 100% de los casos.
* Con una precisión de 1.0, significa que todas las predicciones positivas del modelo fueron correctas.
* Con un recall de 1.0, significa que el modelo identificó correctamente todos los casos positivos reales.
* Con un F1 Score de 1.0, el modelo tiene un equilibrio perfecto entre precisión y recall.
* Pruebas de Integración:

Para la integración del API se utilizó un script desarrollado con Python utilizando la biblioteca requests

* + Verificación de la integración entre el modelo, la API y el frontend.

Para el desarrollo de las pruebas se realizó variando los parámetros de ingreso en el script, teniendo como resultado:

* Caso 1:

Para el primer caso se utilizó los datos:

* + 'TIPO\_DE\_DISCAPACIDAD': '4',
  + 'EDAD\_ANIOS\_MESES': 15,
  + 'PORCENTAJE': 0.7

Obteniendo como resultado: 'NO'

* Caso 2:
  + 'TIPO\_DE\_DISCAPACIDAD': '2',
  + 'EDAD\_ANIOS\_MESES': 8,
  + 'PORCENTAJE': 0.3

Obteniendo como resultado: 'SI'

* Caso 3:
  + 'TIPO\_DE\_DISCAPACIDAD': '3',
  + 'EDAD\_ANIOS\_MESES': 17,
  + 'PORCENTAJE': 0.55

Obteniendo como resultado: 'SI'

* Caso 3:
  + 'TIPO\_DE\_DISCAPACIDAD': '10',
  + 'EDAD\_ANIOS\_MESES': 13,
  + 'PORCENTAJE': 0.61

Obteniendo como resultado: 'NO'

9. Conclusiones

* Resumen de Resultados:
  + Principales hallazgos del proyecto.
* El modelo entrenado ha mostrado un desempeño excelente en el conjunto de prueba, con una precisión (accuracy), precisión (precision), recall y F1 Score de 1.0. Esto indica que el modelo ha clasificado correctamente todas las muestras del conjunto de prueba, sugiriendo que el modelo ha aprendido a distinguir perfectamente entre las clases en los datos disponibles.
* La API creada con Flask está funcionando correctamente, manejando solicitudes POST y devolviendo predicciones precisas. Las pruebas realizadas confirmaron que la integración entre el modelo y la API es robusta, con respuestas precisas a los datos enviados.
* Las pruebas de integración con el script en Python demostraron que la comunicación entre el frontend y la API se realiza de manera efectiva, con resultados consistentes para diferentes entradas.
* Recomendaciones:
* Ampliar el conjunto de datos con más ejemplos y clases puede ayudar a mejorar la generalización del modelo y evitar el sobreajuste.
* Revisar y posiblemente incluir características adicionales o más relevantes que puedan mejorar el rendimiento del modelo.
* Implementar técnicas de validación cruzada para obtener una estimación más robusta del rendimiento del modelo en diferentes particiones de datos.
* Mejorar el manejo de datos inválidos o incompletos para que la API sea más robusta frente a entradas erróneas o mal formateadas.
* Desarrollar una interfaz de usuario más intuitiva y amigable, posiblemente utilizando frameworks modernos como React, Vue o Angular para una mejor experiencia de usuario.

10. Referencias

* Fuentes Citadas:
  + Pallets Projects. (n.d.). Flask Documentation. En Flask (1.1.x). Recuperado el 8 de agosto de 2024, de https://flask.palletsprojects.com/en/1.1.x/
  + Joblib Contributors. (n.d.). Joblib Documentation. En Joblib 1.0. Recuperado el 8 de agosto de 2024, de https://joblib.readthedocs.io/en/latest/
  + Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., ... & Oliphant, T. E. (2020). Array programming with NumPy. Nature, 585(7825), 357-362. Recuperado de https://numpy.org/doc/stable/

13. Apéndices

* Código Fuente:
  + Fragmentos relevantes del código utilizado en el proyecto.
* Modelo de predicción

# Mostrar los valores únicos y su correspondencia con los números factorized

label\_mapping = dict(enumerate(unique\_labels))

print("Mapping de valores factorized:")

print(label\_mapping)

# Manejar valores faltantes

data\_cleaned.loc[:, 'TIPO DE DISCAPACIDAD '] = data\_cleaned['TIPO DE DISCAPACIDAD '].fillna(data\_cleaned['TIPO DE DISCAPACIDAD '].mode()[0])

data\_cleaned.loc[:, 'EDAD AÑOS/MESES'] = data\_cleaned['EDAD AÑOS/MESES'].fillna(data\_cleaned['EDAD AÑOS/MESES'].median())

data\_cleaned.loc[:, 'PORCENTAJE\n%'] = data\_cleaned['PORCENTAJE\n%'].fillna(data\_cleaned['PORCENTAJE\n%'].median())

# Si los datos son adecuados, entrenar el modelo de Random Forest

if len(distribution\_full) > 1 and len(distribution\_train) > 1 and len(distribution\_test) > 1:

model = RandomForestClassifier(n\_estimators=100, random\_state=42)

model.fit(X\_train, y\_train)

# Guardar el modelo entrenado

joblib.dump(model, 'modelo\_entrenado.pkl')

# Realizar predicciones

y\_pred = model.predict(X\_test)

# Evaluar el rendimiento del modelo

accuracy = accuracy\_score(y\_test, y\_pred)

precision = precision\_score(y\_test, y\_pred, average='macro', zero\_division=1)

recall = recall\_score(y\_test, y\_pred, average='macro', zero\_division=1)

f1 = f1\_score(y\_test, y\_pred, average='macro', zero\_division=1)

conf\_matrix = confusion\_matrix(y\_test, y\_pred)

print(f'Accuracy: {accuracy}')

print(f'Precision: {precision}')

print(f'Recall: {recall}')

print(f'F1 Score: {f1}')

print('Confusion Matrix:')

print(conf\_matrix)

* API

@app.route('/predict', methods=['POST'])

def predict():

try:

data = request.get\_json(force=True)

features = np.array([

data['TIPO\_DE\_DISCAPACIDAD'],

data['EDAD\_ANIOS\_MESES'],

data['PORCENTAJE']

]).reshape(1, -1)

prediction = model.predict(features)

return jsonify(prediction=int(prediction[0]))

except Exception as e:

return jsonify(error=str(e)), 400

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug=True)

* Script Front End

# Enviar solicitud POST a la API

response = requests.post(url, json=data)

# Manejar la respuesta

if response.status\_code == 200:

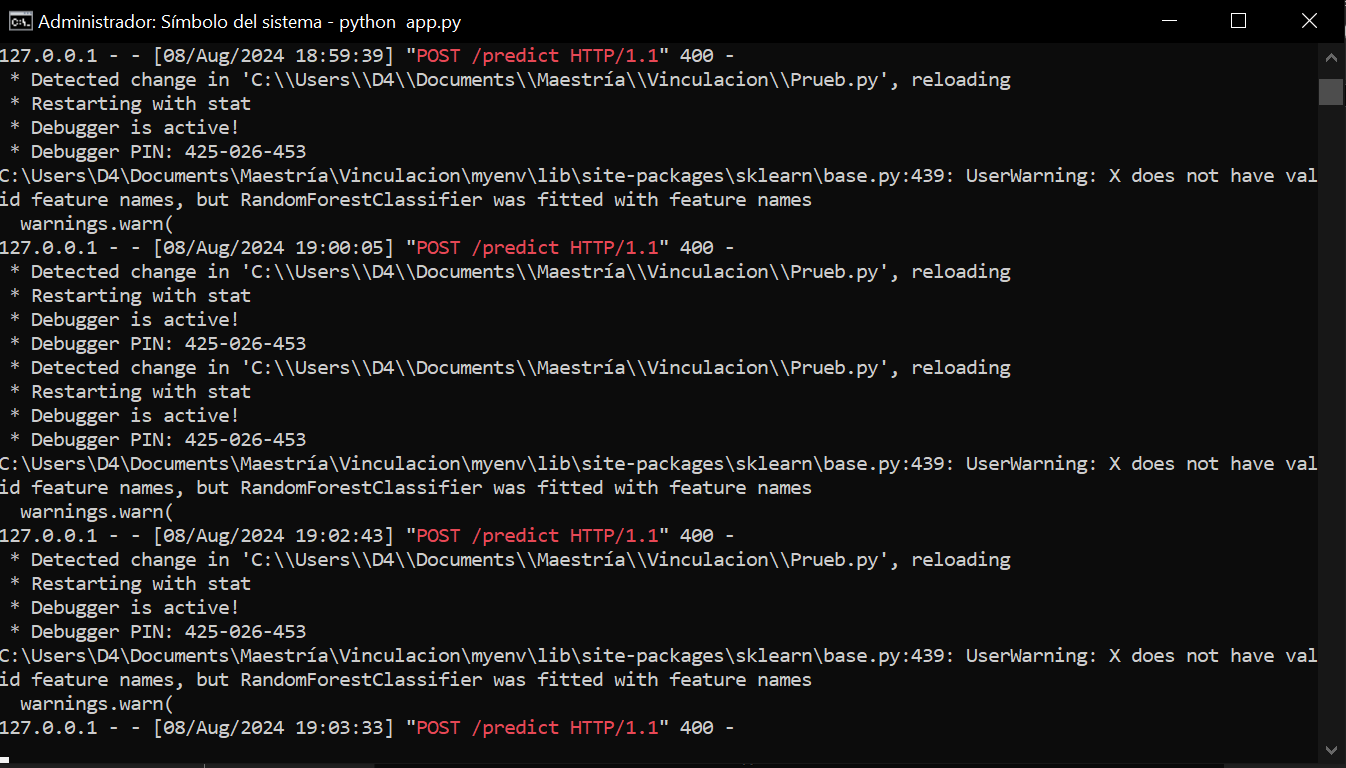
prediction = response.json()['prediction']

print(f'Predicción: {prediction}')

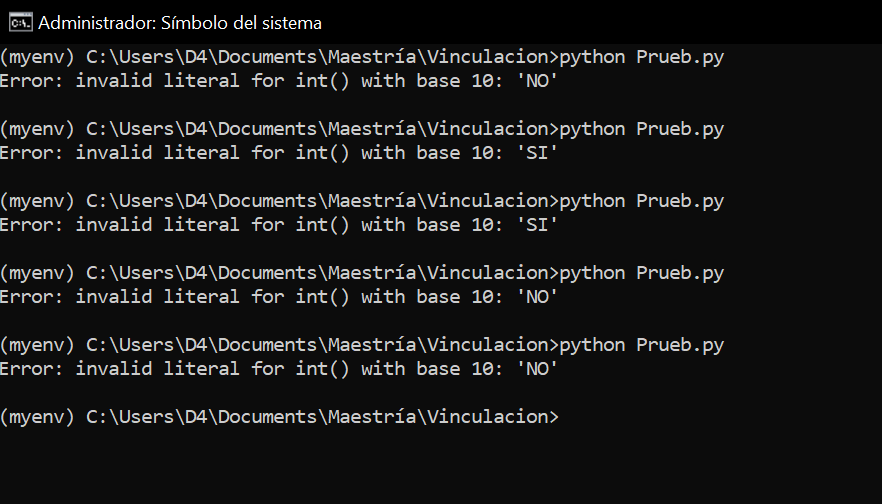
else:

print(f'Error: {response.json()["error"]}')

* Datos Adicionales:
  + Datos de ejemplo, resultados adicionales, y gráficos que no se incluyeron en el cuerpo principal del informe.



API ejecutándose correctamente



Resultado pruebas de integración API con Front End