Tab 1

python joblib library explain

The joblib library in Python is a powerful tool designed for **efficient serialization (saving) and deserialization (loading)** of Python objects, especially large data like **NumPy arrays**, **pandas DataFrames**, and **machine learning models**. It is commonly used in **data science and machine learning pipelines**.

## **🔧 What is joblib?**

joblib is optimized for performance and works well with **large numerical data**, unlike the built-in pickle module which can be slow and inefficient for large arrays.

## **📦 Installing Joblib**

pip install joblib

## **🧠 Key Features**

| **Feature** | **Description** |
| --- | --- |
| **Fast object serialization** | Saves Python objects to disk faster than pickle, especially for large NumPy arrays. |
| **Compression support** | Can compress files to save space using compress=True or specify a level. |
| **Caching** | Memorizes function outputs (results caching) for expensive computations. |
| **Parallel computing** | Easily parallelizes loops using Parallel and delayed. |

## **✅ Common Use Cases**

### **1. Saving and Loading Models**

from joblib import dump, load

from sklearn.ensemble import RandomForestClassifier

model = RandomForestClassifier()

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

# Save model

dump(model, 'rf\_model.joblib')

# Load model

loaded\_model = load('rf\_model.joblib')

### **2. Caching Expensive Function Results**

from joblib import Memory

memory = Memory(location='cache\_dir', verbose=0)

@memory.cache

def slow\_function(x):

import time

time.sleep(5) # Simulate slow computation

return x \* x

slow\_function(10) # Takes 5 sec the first time

slow\_function(10) # Returns instantly from cache

### **3. Parallel Computing**

from joblib import Parallel, delayed

def square(n):

return n \* n

results = Parallel(n\_jobs=4)(delayed(square)(i) for i in range(10))

print(results)

## **🔍 When to Use joblib Over pickle**

| **Use Case** | **Use joblib?** | **Why?** |
| --- | --- | --- |
| Large NumPy arrays | ✅ | Faster and more space-efficient |
| Small Python objects | ❌ | pickle is fine |
| Machine learning models | ✅ | Often includes large data arrays |

## **🧹 File Format**

Joblib stores objects in a directory with binary .pkl or .joblib files (sometimes compressed with .gz or .xz).

## **Summary**

* Use joblib to efficiently **save/load large data structures or ML models**.
* It’s especially useful in **data science workflows** where performance matters.
* Offers **parallel processing**, **result caching**, and **compression** options.

Would you like to see a real-world example with scikit-learn or caching?

python optuna package

Certainly! Here's the explanation of the **Optuna** Python package in English:

## **🔧 What is Optuna?**

**Optuna** is an **automatic hyperparameter optimization framework** for Python. It helps you **tune machine learning models’ hyperparameters efficiently and intelligently**, especially when the search space is large.

It uses a **"define-by-run"** approach, which makes it **flexible, dynamic, and memory-efficient**. That means you define the hyperparameter search space dynamically as your code runs.

## **📦 Installation**

pip install optuna

## **✅ Where is it useful?**

* Hyperparameter tuning (e.g. for XGBoost, LightGBM, CatBoost, etc.)
* Deep learning (e.g. PyTorch, TensorFlow)
* Algorithm or model selection
* Feature selection tasks

## **🔁 Simple Example**

import optuna

from sklearn.datasets import load\_iris

from sklearn.ensemble import RandomForestClassifier

from sklearn.model\_selection import cross\_val\_score

# Load data

X, y = load\_iris(return\_X\_y=True)

# Define objective function

def objective(trial):

n\_estimators = trial.suggest\_int("n\_estimators", 10, 100)

max\_depth = trial.suggest\_int("max\_depth", 2, 32)

clf = RandomForestClassifier(n\_estimators=n\_estimators, max\_depth=max\_depth)

return cross\_val\_score(clf, X, y, cv=3).mean()

# Create and run the study

study = optuna.create\_study(direction="maximize")

study.optimize(objective, n\_trials=50)

# Show best result

print("Best trial:")

print(study.best\_trial)

## **📊 Key Features**

| **Feature** | **Description** |
| --- | --- |
| **Define-by-run API** | Define search space at runtime – dynamic and flexible |
| **Pruning** | Stops unpromising trials early to save time |
| **Visualization** | Built-in charts to track optimization progress |
| **Multi-objective support** | Optimize more than one target metric at the same time |

## **📈 Visualization Example**

optuna.visualization.plot\_optimization\_history(study).show()

optuna.visualization.plot\_param\_importances(study).show()

## **🧠 Did You Know?**

* Optuna integrates easily with **LightGBM, XGBoost, PyTorch, Keras**, and others.
* Trials can be saved to a **SQLite database**, so you can **resume** optimization later.

## **🔚 Summary**

* **Optuna** is a **smart and flexible hyperparameter tuning tool**.
* Automatically finds the best parameters for your model.
* Offers **pruning**, **visualization**, and a **powerful API**.

Would you like me to show how Optuna works with a specific library like **XGBoost** or **PyTorch**?

Keras Input

LSTM layers in Keras require 3D input: (samples, time steps, features) what is this terms

Great question! Let’s break down the terms involved in the **3D input format** for **LSTM layers in Keras**:

## **📦 Format Required:**

### **(samples, time steps, features)**

This is the shape of the input data you give to an LSTM layer. Here's what each dimension means:

### **1. samples**

* The number of sequences or examples in your dataset.
* Each sample is an independent training example.
* 📌 This is the **number of rows** in your training set.

🧠 **Example:** If you're training on 100 time series sequences → samples = 100

### **2. time steps**

* The number of time points (lags) in each input sequence.
* This defines **how much past information** the model can "look back" at for making predictions.

🧠 **Example:** If you're using a **window of 12 past days** to predict the next value → time steps = 12

### **3. features**

* The number of features (variables) at each time step.
* This could be 1 (if you're using just one time series like temperature), or more if you use multiple inputs (e.g., temperature, rainfall, humidity).

🧠 **Example:**

* If you're using only dengue case counts: features = 1
* If you're using 3 variables (e.g., cases, temperature, humidity): features = 3

## **✅ Example in Real Terms:**

Let’s say you’re forecasting dengue cases using the past 10 days of data (time steps = 10) and using 2 features: temperature and rainfall (features = 2), and you have 1,000 training samples:

X\_train.shape = (1000, 10, 2)

* 1000 samples
* Each sample contains 10 time steps
* Each time step has 2 features

### **📌 Summary Table:**

| **Dimension** | **Description** | **Example Value** |
| --- | --- | --- |
| samples | Number of training sequences | 1000 |
| time steps | Length of each input sequence (past steps) | 10 |
| features | Number of variables at each time step | 2 |

Let me know if you want a visual diagram or sample code to generate this shape using pandas or NumPy.

**if X\_train.ndim < 3: X\_train = X\_train.reshape(X\_train.shape[0], X\_train.shape[1], 1) X\_test = X\_test.reshape(X\_test.shape[0], X\_test.shape[1], 1)**

This code ensures that your **training and test data** are in the correct shape for an **LSTM model in Keras**, which **requires 3D input**.

### **🔍 Code:**

if X\_train.ndim < 3:

X\_train = X\_train.reshape(X\_train.shape[0], X\_train.shape[1], 1)

X\_test = X\_test.reshape(X\_test.shape[0], X\_test.shape[1], 1)

### **🧠 What It Does:**

* Checks if X\_train has **less than 3 dimensions** (i.e., likely 2D, shape = (samples, time\_steps)).

If true, it reshapes X\_train and X\_test into 3D format:  
  
 (samples, time\_steps, features)

### **🧪 Example:**

Let’s say you have:

X\_train.shape = (500, 12)

X\_test.shape = (100, 12)

➡️ This means you have:

* 500 training samples
* Each sample uses a window of 12 time steps
* But only **1 feature** (probably a univariate time series)

After reshaping:

X\_train.shape → (500, 12, 1)

X\_test.shape → (100, 12, 1)

Now each time step has 1 feature, which is required by the LSTM.

### **✅ Why It's Needed:**

Keras LSTM expects input in this shape:

(batch\_size, timesteps, features)

If your data is only 2D, it assumes you're missing the features dimension (e.g., only 1 variable). So you add it explicitly by reshaping.

### **🟡 Caution:**

* This code assumes you're working with **univariate time series**.  
   If you already have multiple features (e.g., (500, 12, 3)), reshaping is not needed — and could even **break** the shape.

Let me know if you'd like a reusable function to handle this reshaping safely!