Machine Learning

ML with Python: Setup & Introduction

Gabriele Russo Russo   Francesco Lo Presti

Laurea Magistrale in Ingegneria Informatica – A.Y. 2023/24

TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

Dipartimento di Ingegneria Civile e Ingegneria Informatica

Outline

- Environment setup
- NumPy & Pandas: Basics
- Scikit-learn and ML workflow example

Environment Setup

Requirements

We will use Python as the reference programming language for the course.

Software requirements:

- Python 3.x
- Some libraries, e.g.:
  - `scikit-learn`
  - `tensorflow`
- Jupyter (Lab)

Jupyter Notebooks

- **Jupyter Notebook**: web application for creating computational documents (i,e., notebooks), which may contain Python code, text, figures
- **Jupyter Lab**: evolution of the original web app, providing a development environment to work with notebooks
- Pick the one you prefer

Setup Methods

- Method 0: naive approach (not recommended)
- Method 1: venv
- Method 2: conda
- Method 3: Google Colab

Method 0 (Not recommended)

- Install Python and Jupyter following the instructions for your OS
- Install the libraries you need using `pip`
  - e.g., `pip install scikit-learn`
- Difficult to install specific library versions, if required
- System upgrades can break your project

Isolated Environments

- Isolated Python environments avoid the issue of conflicting Python/library versions across different projects
- Libraries installed within an environment not visible outside

- Option 1: Lightweight environments
  - A directory with all the install libraries
  - Default Python interpreter of the system is used
- Option 2: `conda` environments
  - Libraries + Python interpreter

Method 1: venv

- Install Python following the instructions for your OS
- Create a virtual environment using `venv`
- Install the libraries you need using `pip`
- Jupyter can be either installed at system-level or using `pip` in the env.
- No third-party tools to install
- Same Python version across environments
- Python upgrades can break your environments

Method 1: Example

```
# Create environment (in the current directory)
python -m venv my_env
# Activate it
source ./my_env/bin/activate
# Install libraries...
pip install scikit-learn
# ...
deactivate
```

Requirements file

- It is a good practice to list the required libraries (and their version) in a `requirements.txt` file

Example requirements.txt

```
matplotlib==3.6.1
numpy==1.23.4
packaging==21.3
```

- You can easily reinstall them with a single command

```
pip install -r requirements.txt
```

Method 2: conda

- Install `conda`, picking one of the available distributions, e.g.:
  - `miniforge`
  - `miniconda`
  - ...
- Create an environment using `conda`
- Install libraries and Jupyter using `conda install`
- Each environment can use its own Python version and libraries

Method 2: Example

```
# Create environment
conda create -n ENVNAME python=3.10
# Activate it
conda activate ENVNAME
# Install stuff
conda install jupyterlab
conda install scikit-learn

jupyter lab

conda deactivate
```

Note: base environment

- When installing a conda distribution, a base environment is created
- By default, the base environment is activated automatically when no other env is in use
- If you want to disable this behavior (recommended):

```
conda config --set auto_activate_base false
```

Importing/exporting environments

```
# Export to file
conda env export > ENV.yml
# Cross-platform export
conda env export --from-history > ENV.yml

# Import
conda env create -n ENVNAME --file ENV.yml
```

We will share a `env.yml` file that you can import

Method 3: Google Colab

- Google Colaboratory (or, Colab) is a cloud-based platform to run Jupyter notebooks in your browser
- Usage (with some limitations) is free and only requires a Google account
- Colab can connect to Google Drive and GitHub to load notebooks
- No need to install anything on your PC, except for a browser
- Effortless access to GPUs
- `https://colab.research.google.com`

NumPy & Pandas: Basics

NumPy

NumPy (Numerical Python) provides data structures and functions to efficiently work on large N-dimensional numerical arrays. Key features:

- **ndarray** (N-dimensional array)
- functions to operate on ndarrays
- I/O functions

NumPy is conventionally imported as follows:

```python
import numpy as np
```

ndarray

- N-dimensional array of numbers (either integers or floats)
- Support for efficient and easy manipulation (you can work on vectors and matrices without loops)

```
import numpy as np
A = np.array([1,2,5])
print(A)
print(A*2)
```

Creating a ndarray

Converting from a different collection (e.g., list)

```
A = np.array([1,2,3,4])
M = np.array([[1,2,3], [7,8,9]])
```

Specifying arbitrary dimensions

```
A = np.zeros((5,3)) # not zeros(5,3)!
B = np.ones(15)
```

From file

```
A = np.loadtxt("filename.txt")
```

Basic operations

```
A = np.ones((2,2))
B = np.array([[1,2],[0,1]])
print(A+B)
print(A+5*B)
print(np.matmul(A,B.transpose()))
```

shape

- The shape field is a tuple indicating the dimensions

```
A = np.array([[1,2], [4,5]])
print(A.shape) # (2,2)
```

- Sometimes, it is useful to “reshape” a ndarray. For instance, if you have a 1-D array of length N, you may want to reshape it to a 1xN matrix.

```
A = np.array([1,2,3])
B = np.array([[1,2,3]])
print(np.matmul(A.T,B))  # error
A = A.reshape(1,3)
print(np.matmul(A.T,B)) # 3x3 matrix
```

Indexing and slicing

You can easily access single elements or slices:

```
print(A[4])
print(A[4:6])
A[4:6] = -1.0
A[:] = 5  # assign 5 to all elements
# For a matrix:
print(M[1,2])  # or M[1][2]
print(M[:1,:])
```

References

- Official docs:
  `https://numpy.org/doc/stable/user/index.html`

Pandas

Pandas is a Python library providing data structures and functions to efficiently manipulate data. Key data structures:

- **Series**
- **DataFrame**

Differently from NumPy, Pandas can handle tabular data with heterogeneous data types (i.e., not limited to numerical values).

```
import pandas as pd
```

Series

- Series is a one-dimensional labeled array capable of holding any data type (integers, strings, floating point numbers, Python objects, etc.).
- The axis labels are collectively referred to as the index
  - e.g., in a time series, values might be labeled with a date
  - If not specified, the default index consists of integers from 0 to $(N-1)$ (where N is the number of values)
- Similar to NumPy's ndarray, but with labels
  - you can pass a Series to many functions acting on ndarrays

```
A = pd.Series([4, 7, 5], index=["a", "b", "c"])
print(A)
```

DataFrame

- DataFrame is a 2-dimensional labeled data structure with columns of potentially different types.
- You can think of a DataFrame as a dictionary of Series objects (each representing a column)
- Both rows and columns have a corresponding array of labels: index for rows, columns for columns

```
df = {"one": [1.0, 2.0, 3.0, 4.0],
      "two": [4.0, 3.0, 2.0, 1.0]}
pd.DataFrame(df)
#    one  two
#0   1.0  4.0
#1   2.0  3.0
#...
```

Value Selection

- Two fundamental functions to select values based on their integer location (`.iloc()`) or their label (`.loc()`)

```
df.columnName # -> Series
df["columnName"]  # -> Series
df.loc[:, "columnName"] # -> Series
df.loc[:, ["colA", "colB"]] # -> DataFrame

df.iloc[3, :] # -> Series
df.iloc[3, 2:5] # -> DataFrame

df.loc[bool_vector] # -> DataFrame
df.loc[df.columnName > 3]
```

Reading a CSV File

```
df = pd.read_csv("nomefile.csv")
```

- By default, first line is used to extract column names, unless `header=None` is specified
- `skiprows=N` allows to skip first N lines in the file
- `names=["a",...]` allows manual column naming
- `index_col="a"` specifies the column to use as index

scikit-learn and ML workflow

Scikit-learn

- Scikit-learn is an open source ML library that supports supervised and unsupervised learning
- It also provides various tools for model fitting, data preprocessing, model selection, model evaluation, and many other utilities
- `https://scikit-learn.org/`
- We will see scikit-learn in action by presenting an end-to-end example ML project

Scikit-learn: Key Concepts

- Scikit-learn provides dozens of built-in algorithms and models, called estimators, which implement a common interface
- Each estimator can be trained against some data using its fit() method
- Once the estimator is fitted, it can be used for predicting target values of new data, using the predict() method
- Data are provided to scikit-learn as array-like objects, usually consisting of Numpy's ndarrays or Pandas Series/DataFrame

End-to-End ML Workflow Example

- We consider an example project, described in greater detail in Chapter 2 of “Hands-on ML with Scikit-learn, ...”
- Our task is predicting median house prices in California districts, based on the well-known *California Housing Prices* dataset
  - Data from 1990 California census
  - Actually, we will use a slightly simplified version

Understanding the Task

- As a ML engineer, your first step is understanding the business needs and goals behind a project
- For instance, we are requested to predict house prices. How will be the prediction used?
  - e.g., to help potential buyers estimate prices
  - e.g., to help investors deciding whether to buy or not
- This allows you to understand how critical the task is and choose a suitable performance metric
  - e.g., in a classification task, are false positives less harmful than false negatives?

Understanding the Task

- As a ML engineer, your first step is understanding the business needs and goals behind a project
- For instance, we are requested to predict house prices. How will be the prediction used?
  - e.g., to help potential buyers estimate prices
  - e.g., to help investors deciding whether to buy or not
- This allows you to understand how critical the task is and choose a suitable performance metric
  - e.g., in a classification task, are false positives less harmful than false negatives?
- We use a typical performance measure for regression

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(t^{(i)} - y^{(i)}\right)^2}$$

Hands-on Example

The example is presented in a Jupyter notebook.

end2end.ipynb