# The Python Toolbox for Machine Learning (without PyTorch / TensorFlow / OpenCV)

Your Name

Course / Lab

January 21, 2026

## Scope and goal

- We **do not** cover: PyTorch, TensorFlow, OpenCV
- We **do** cover the tools that make model-building easier later:
  - arrays + linear algebra
  - data handling and cleaning
  - visualization and debugging
  - classical ML utilities
  - experiment hygiene (reproducibility, logging, configs)
  - performance and profiling
- Goal: when we later teach deep learning, students already speak the language.

## Roadmap

1. Environment and project structure
2. NumPy: arrays, broadcasting, vectorization
3. SciPy: scientific building blocks (optimize, stats)
4. Pandas: tables, joins, missing values
5. Visualization: Matplotlib (+ Seaborn optional)
6. scikit-learn: preprocessing, splitting, metrics, pipelines
7. Data sources: CSV/JSON/Parquet, images (Pillow), datasets (huggingface)
8. Experiment hygiene: logging, argparse, pathlib, tqdm, random seeds
9. Performance tools: profiling, numba (optional)

## Recommended environment tools

- **pip + venv** or **conda/mamba**: isolated environments
- **Jupyter** for exploration, **.py scripts** for reproducible runs
- **Version control (git)** from day 1
- Basic rule: *notebooks discover, scripts deliver*

## A minimal ML project layout

```
project/
  data/               # raw / processed data (often git-ignored)
  notebooks/          # exploration
  src/
    dataset.py
    features.py
    train.py
    eval.py
    utils.py
  tests/
  requirements.txt
  README.md
```

- Keep reusable code in `src/`, not hidden in notebooks.

## NumPy is the mental model

- Almost every ML library assumes you understand:
    - arrays and shapes
    - broadcasting
    - vectorized operations
    - linear algebra: matrix multiply, norms, SVD
- Deep learning tensors behave like NumPy arrays (plus gradients).

## Creating arrays and inspecting them

```python
import numpy as np

x = np.array([1, 2, 3], dtype=np.float32)
A = np.random.randn(4, 3)          # shape (4,3)
Z = np.zeros((2, 5))
I = np.eye(3)

print(A.shape, A.dtype)            # (4,3) float64
print(A.ndim, A.size)              # 2, 12
```

## Indexing, slicing, boolean masks

```python
import numpy as np

A = np.arange(12).reshape(3, 4)
print(A[0, 2])          # element
print(A[:, 1])          # column
print(A[1:3, 2:])       # submatrix

mask = A % 2 == 0       # even entries
evens = A[mask]         # 1D array of selected elements
A[mask] = -1            # in-place assignment
```

## Broadcasting: the shape superpower

- Broadcasting aligns shapes from the **right**
- A dimension of size 1 can be expanded
- Typical ML pattern: normalize features

$$X \in \mathbb{R}^{N \times D},\ \mu \in \mathbb{R}^{D},\ \sigma \in \mathbb{R}^{D} \Rightarrow \hat{X} = \frac{X - \mu}{\sigma}$$

## Broadcasting example: feature normalization

```
import numpy as np

X = np.random.randn(1000, 5)                # (N,D)
mu = X.mean(axis=0)                         # (D,)
std = X.std(axis=0) + 1e-8                  # (D,)
Xn = (X - mu) / std                         # broadcasting

print(Xn.mean(axis=0))                      # ~0
print(Xn.std(axis=0))                       # ~1
```

## Vectorization vs loops (why ML code is fast)

```python
import numpy as np

# Compute pairwise squared distances between rows of X and Y
X = np.random.randn(1000, 64)      # (N,D)
Y = np.random.randn(500, 64)       # (M,D)

# Vectorized:
# ||x-y||^2 = ||x||^2 + ||y||^2 - 2 x y
X2 = (X**2).sum(axis=1, keepdims=True)  # (N,1)
Y2 = (Y**2).sum(axis=1, keepdims=True)  # (M,1)
D2 = X2 + Y2.T - 2 * (X @ Y.T)          # (N,M)

print(D2.shape)  # (1000, 500)
```

## Linear algebra essentials

```
import numpy as np

A = np.random.randn(5, 3)
b = np.random.randn(5)

# Least squares: minimize ||Ax - b||^2
x_hat, *_ = np.linalg.lstsq(A, b, rcond=None)

# SVD: A = U S V^T
U, S, Vt = np.linalg.svd(A, full_matrices=False)

# Norms
l2 = np.linalg.norm(A)             # Frobenius norm for matrices
```

## NumPy habits that pay dividends later

- Track shapes like a hawk: `(N, D)` vs `(D,)` vs `(N,1)`
- Prefer `keepdims=True` when you want stable broadcasting
- Avoid Python loops over samples when possible
- Use `np.random.default_rng(seed)` for reproducible randomness

## SciPy: the toolbox behind the toolbox

- SciPy provides reliable implementations for:
    - optimization (`scipy.optimize`)
    - statistics (`scipy.stats`)
    - sparse matrices (`scipy.sparse`)
    - distances (`scipy.spatial`)
- Useful for baselines, classical methods, and sanity checks.

## Optimization example: fit logistic regression by hand

```
import numpy as np
from scipy.optimize import minimize

rng = np.random.default_rng(0)
X = rng.normal(size=(200, 3))
true_w = np.array([1.5, -2.0, 0.5])
logits = X @ true_w
y = (logits + 0.2 * rng.normal(size=200) > 0).astype(np.float64)

def sigmoid(z): return 1 / (1 + np.exp(-z))

def nll(w):
    z = X @ w
    p = sigmoid(z)
    eps = 1e-9
    return -(y*np.log(p+eps) + (1-y)*np.log(1-p+eps)).mean()

res = minimize(nll, x0=np.zeros(3), method="BFGS")
```

## Statistics example: confidence intervals by bootstrap

```python
import numpy as np

rng = np.random.default_rng(0)
data = rng.normal(loc=2.0, scale=1.0, size=200)

B = 2000
means = []
for _ in range(B):
    sample = rng.choice(data, size=len(data), replace=True)
    means.append(sample.mean())
means = np.array(means)

ci = np.quantile(means, [0.025, 0.975])
print("mean:", data.mean(), "95% CI:", ci)
```

## Pandas: tables, not tensors

- Pandas is for:
  - loading datasets (CSV/Parquet)
  - cleaning, merging, feature engineering
  - exploring distributions and missing values
- Typical workflow: Pandas for **ETL**, NumPy for **math**.

## Loading data and basic inspection

```python
import pandas as pd

df = pd.read_csv("data/train.csv")    # or read_parquet(...)
print(df.head())
print(df.dtypes)
print(df.isna().mean().sort_values(ascending=False).head(10))
```

## Filtering, selecting, and creating features

```python
import pandas as pd
import numpy as np

df = pd.DataFrame({
    "age": [20, 35, 40, None],
    "income": [30_000, 80_000, 50_000, 60_000],
    "city": ["A", "B", "A", "C"]
})

# Filter rows
adults = df[df["age"].fillna(0) >= 18]

# Create a feature
df["log_income"] = np.log(df["income"])

# Column selection
X = df[["age", "log_income", "city"]]
```

## Groupby: aggregation and statistics

```python
import pandas as pd

df = pd.DataFrame({
    "city": ["A", "A", "B", "B", "B"],
    "y":    [1, 0, 1, 1, 0],
    "x":    [0.2, 0.1, 0.9, 0.7, 0.4]
})

stats = df.groupby("city").agg(
    n=("y", "size"),
    y_mean=("y", "mean"),
    x_mean=("x", "mean")
).reset_index()

print(stats)
```

## Merging datasets (joins)

```
import pandas as pd

users = pd.DataFrame({"user_id":[1,2,3], "country":["FR","CH","FR"
    ]})
events = pd.DataFrame({"user_id":[1,1,2], "event":["click","buy","
    click"]})

df = users.merge(events, on="user_id", how="left")
print(df)
```

## Missing values: strategies

```python
import pandas as pd

# 1) Drop rows (risky if many)
df_drop = df.dropna()

# 2) Impute numeric with median
df["age"] = df["age"].fillna(df["age"].median())

# 3) Add missingness indicator
df["age_missing"] = df["age"].isna().astype(int)
```

## Why plots matter in ML

- Plots are debugging tools wearing a nice hat.
- Use them to catch:
  - data leakage
  - weird distributions / outliers
  - class imbalance
  - training instability (loss spikes)

## Matplotlib: minimal plotting pattern

```python
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 10, 200)
y = np.sin(x)

plt.figure()
plt.plot(x, y, label="sin(x)")
plt.xlabel("x")
plt.ylabel("y")
plt.title("A simple plot")
plt.legend()
plt.show()
```

## Histogram + class imbalance check

```python
import matplotlib.pyplot as plt
import numpy as np

y = np.random.choice([0, 1], size=1000, p=[0.9, 0.1])

counts = np.bincount(y)
plt.figure()
plt.bar([0, 1], counts)
plt.xticks([0, 1], ["class 0", "class 1"])
plt.title("Class counts (imbalance?)")
plt.show()
```

## Confusion matrix visualization

```
import numpy as np
import matplotlib.pyplot as plt

cm = np.array([[50,  5],
               [12, 33]])

plt.figure()
plt.imshow(cm)
plt.colorbar()
plt.xticks([0,1], ["pred 0","pred 1"])
plt.yticks([0,1], ["true 0","true 1"])

for i in range(2):
    for j in range(2):
        plt.text(j, i, str(cm[i,j]), ha="center", va="center")

plt.title("Confusion Matrix")
plt.show()
```

## Seaborn (optional): quick statistical plots

```python
import seaborn as sns
import pandas as pd
import numpy as np

df = pd.DataFrame({
    "feature": np.random.randn(500),
    "label": np.random.choice([0, 1], size=500)
})

sns.kdeplot(data=df, x="feature", hue="label", common_norm=False)
```

## scikit-learn: what you must know before deep learning

- Data splitting: train/val/test
- Preprocessing: scaling, one-hot encoding, imputation
- Metrics: accuracy, F1, ROC-AUC, confusion matrix
- Pipelines: avoid leakage by bundling steps
- Baselines: always compare against something simple

## Train/val split + scaling + baseline model

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

rng = np.random.default_rng(0)
X = rng.normal(size=(1000, 20))
w = rng.normal(size=20)
y = (X @ w + 0.5*rng.normal(size=1000) > 0).astype(int)

X_train, X_val, y_train, y_val = train_test_split(
    X, y, test_size=0.2, random_state=0, stratify=y
)

scaler = StandardScaler()
X_train_s = scaler.fit_transform(X_train)
X_val_s = scaler.transform(X_val)
```

## The leakage trap (and how pipelines save you)

- Wrong:
  - scale **all** data, then split (leaks information)
- Right:
  - split, then fit scaler on train only
  - or use a **Pipeline** (recommended)

## Pipeline with preprocessing + model

```python
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression

pipe = Pipeline([
    ("scaler", StandardScaler()),
    ("clf", LogisticRegression(max_iter=200))
])

pipe.fit(X_train, y_train)
pred = pipe.predict(X_val)
```

## Categorical features: OneHotEncoder + ColumnTransformer

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression

df = pd.DataFrame({
    "age": [20, 35, 40, 22, 19],
    "city": ["A", "B", "A", "C", "B"],
    "y": [0, 1, 1, 0, 0]
})

X = df[["age", "city"]]
y = df["y"]

num_cols = ["age"]
```

## Metrics you should teach early

```python
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score, f1_score,
    confusion_matrix, classification_report
)

y_true = [0, 1, 1, 0, 1]
y_pred = [0, 1, 0, 0, 1]

print("acc:", accuracy_score(y_true, y_pred))
print("f1 :", f1_score(y_true, y_pred))
print("cm :\n", confusion_matrix(y_true, y_pred))
print(classification_report(y_true, y_pred))
```

## Cross-validation and hyperparameter search

```python
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC

param_grid = {"C": [0.1, 1, 10], "gamma": ["scale", 0.1, 1.0]}
search = GridSearchCV(SVC(), param_grid, cv=5, scoring="f1")
search.fit(X_train, y_train)

print("best params:", search.best_params_)
print("best score :", search.best_score_)
```

## File formats: choose wisely

- **CSV**: universal, but slow and type-unsafe
- **JSON**: good for nested metadata, not for large tables
- **Parquet**: fast, compressed, typed (great for big tabular data)
- **NumPy .npz**: fast arrays for experiments

## Pandas: CSV vs Parquet

```python
import pandas as pd

df = pd.read_csv("data/train.csv")
df.to_parquet("data/train.parquet", index=False)

df2 = pd.read_parquet("data/train.parquet")
```

## JSON for metadata

```python
import json
from pathlib import Path

meta = {"dataset": "cats_vs_dogs", "n": 25000, "classes": ["cat", "
    dog"]}
Path("meta.json").write_text(json.dumps(meta, indent=2))

loaded = json.loads(Path("meta.json").read_text())
print(loaded["classes"])
```

## Images without OpenCV: Pillow (PIL)

- Pillow lets you:
  - load/save common formats (PNG/JPEG)
  - resize/crop
  - convert to NumPy arrays for feature extraction or visualization
- Later, deep learning pipelines often start from Pillow images.

## Pillow example: load, resize, to NumPy

```python
from PIL import Image
import numpy as np

img = Image.open("data/img001.jpg").convert("RGB")
img_small = img.resize((128, 128))

arr = np.asarray(img_small)          # shape (H,W,C), dtype uint8
arr_f = arr.astype(np.float32) / 255 # normalize to [0,1]

print(arr.shape, arr_f.min(), arr_f.max())
```

## Visualize an image with matplotlib

```python
import matplotlib.pyplot as plt
from PIL import Image

img = Image.open("data/img001.jpg").convert("RGB")
plt.figure()
plt.imshow(img)
plt.axis("off")
plt.title("Sample image")
plt.show()
```

## Datasets in practice: Hugging Face Datasets (optional)

- Convenient access to many public datasets
- Provides:
  - lazy loading and streaming
  - standardized train/test splits
  - dataset filtering and mapping
- Great for teaching, prototyping, and quick baselines.

## Hugging Face datasets: quick taste

```python
from datasets import load_dataset

ds = load_dataset("imdb")                # sentiment dataset
print(ds["train"][0])

# map: create new column
def length(example):
    example["n_chars"] = len(example["text"])
    return example

ds2 = ds["train"].select(range(1000)).map(length)
print(ds2[0]["n_chars"])
```

## Why hygiene matters before deep learning

- Deep models are noisy: you need discipline to know what changed.
- Learn this now:
  - reproducible randomness
  - structured logging
  - clear configs / CLI arguments
  - progress bars

## Randomness: reproducible experiments

```python
import numpy as np
import random

seed = 0
random.seed(seed)
rng = np.random.default_rng(seed)

x1 = rng.normal(size=3)
rng = np.random.default_rng(seed)
x2 = rng.normal(size=3)

print(np.allclose(x1, x2))  # True
```

## pathlib: clean file paths (portable)

```python
from pathlib import Path

data_dir = Path("data")
csv_path = data_dir / "train.csv"

if csv_path.exists():
    print("Found:", csv_path.resolve())
```

## argparse: configurable scripts

```python
import argparse

def main():
    p = argparse.ArgumentParser()
    p.add_argument("--lr", type=float, default=1e-3)
    p.add_argument("--epochs", type=int, default=10)
    args = p.parse_args()

    print("training with lr=", args.lr, "epochs=", args.epochs)

if __name__ == "__main__":
    main()
```

## logging: messages with levels

```python
import logging

logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s | %(levelname)s | %(message)s"
)

logging.info("starting training")
logging.warning("class imbalance detected")
logging.error("failed to load file")
```

## tqdm: progress bars for loops

```python
from tqdm import tqdm
import numpy as np

losses = []
for step in tqdm(range(1000)):
    loss = np.exp(-step / 200) + 0.05*np.random.rand()
    losses.append(loss)
```

## Saving artifacts: metrics, models, plots

```python
import json
from pathlib import Path

out = Path("runs/exp001")
out.mkdir(parents=True, exist_ok=True)

metrics = {"val_acc": 0.91, "val_f1": 0.88}
(out / "metrics.json").write_text(json.dumps(metrics, indent=2))
```

## Config management (optional but powerful)

- For small classes: `argparse` + a config dict is enough
- For larger projects, consider:
  - **pydantic** (validated configs)
  - **omegaconf / hydra** (hierarchical configs)
- Teaching tip: introduce configs once students feel the pain of copy-paste experiments.

## pydantic config example (optional)

```python
from pydantic import BaseModel, Field

class TrainConfig(BaseModel):
    lr: float = Field(1e-3, gt=0)
    epochs: int = Field(10, ge=1)
    batch_size: int = Field(64, ge=1)

cfg = TrainConfig(lr=5e-4, epochs=20, batch_size=128)
print(cfg.model_dump())
```

## Performance: what matters before GPUs

- Bottlenecks often come from:
    - Python loops
    - slow file I/O
    - repeated preprocessing
- Learn to measure first, optimize second.

## timeit: quick measurement

```python
import timeit
import numpy as np

setup = "import numpy as np; X=np.random.randn(10000, 128); w=np.
   random.randn(128)"
stmt = "X @ w"

print(timeit.timeit(stmt, setup=setup, number=200))
```

## cProfile: find slow functions

```python
import cProfile
import pstats

def slow():
    s = 0
    for i in range(1_000_00):
        s += i*i
    return s

prof = cProfile.Profile()
prof.enable()
slow()
prof.disable()

pstats.Stats(prof).sort_stats("cumtime").print_stats(10)
```

## Numba (optional): make numerical loops fast

- **numba** can JIT-compile numerical Python code
- Great for custom feature extraction or simulation loops
- Teaching tip: show once, then emphasize vectorization first

## Numba example (optional)

```python
import numpy as np
from numba import njit

@njit
def pairwise_l2(X):
    n = X.shape[0]
    D = np.zeros((n, n))
    for i in range(n):
        for j in range(n):
            s = 0.0
            for k in range(X.shape[1]):
                d = X[i,k] - X[j,k]
                s += d*d
            D[i,j] = s
    return D
```

## How this maps to deep learning later

- NumPy arrays → deep learning tensors (same shape rules)
- Pandas → dataset cleaning / feature creation
- Matplotlib → debug data + visualize training curves
- scikit-learn → baselines + splits + metrics + preprocessing
- Logging/argparse/pathlib/tqdm → reproducible training scripts
- Pillow/datasets → loading real-world data (images/text)

## A minimal dependency set to recommend

- **Core**: numpy, scipy, pandas, matplotlib
- **ML utilities**: scikit-learn
- **Quality of life**: tqdm
- **Images (non-OpenCV)**: pillow
- **Optional**: seaborn, datasets, pydantic, numba

## Suggested teaching flow

1. NumPy shapes + broadcasting + vectorization (the big unlock)
2. Pandas cleaning + train/val split
3. scikit-learn pipeline + metrics (avoid leakage)
4. Matplotlib: distributions + confusion matrix + learning curves
5. Experiment hygiene: argparse + logging + tqdm + seeds
6. Only then: deep learning framework (students won't drown)

## Questions?

If your array shape looks innocent, it is probably plotting a coup.