Clase 004 — Estructura reproducible de proyecto (cookiecutter-data-science)

Parte: 0 — Prerrequisitos · Fuente: cookiecutter-data-science v2 · Hidden Technical Debt in ML Systems (Sculley et al., 2015). · Duración estimada: 60 min.

Clase 004 — Estructura reproducible de proyecto (cookiecutter-data-science)

> *Parte: 0 — Prerrequisitos · Fuente: cookiecutter-data-science v2 · Hidden Technical Debt in ML Systems (Sculley et al., 2015). · Duración estimada: 60 min.*

Objetivo

Que el alumno deje de crear proyectos como "una carpeta con notebooks" y empiece a usar una estructura estándar que separa código, datos, modelos, notebooks de exploración y documentación. Esto no es estética — es lo que permite que un compañero entienda el proyecto en 5 minutos y que el código viva más allá del notebook donde nació.

Resultados de aprendizaje

Al finalizar la clase, el alumno podrá:

1. Generar un proyecto con la plantilla cookiecutter-data-science (CCDS v2).
2. Justificar la separación data/raw (inmutable) ↔ data/interim ↔ data/processed.
3. Mover código de un notebook a src/ cuando deja de ser exploratorio.
4. Documentar dependencias en pyproject.toml (no en requirements.txt suelto).
5. Reconocer los olores de un proyecto mal estructurado (notebooks con números 01/02/03, código duplicado, datos en git).

Temas

#	Tema	Por qué importa
1	Estructura CCDS v2	Convención > improvisación.
2	data/raw es sagrado	Nunca se modifica; siempre se puede regene
3	notebooks/ vs src/	Exploración vs producción.
4	pyproject.toml como fuente de verdad de de	Reemplaza requirements.txt suelto.
5	Makefile como interfaz	make data, make train, make test — lo lee
6	Olores típicos	Untitled27.ipynb, datos en git, final_FINA

Complemento: Testing con pytest

El folder tests/ que genera CCDS suele quedar vacío — y es un error. Un data scientist necesita tests por tres razones concretas: las funciones de feature engineering se reusan en producción y un bug silencioso te corrompe el dataset; los pipelines reproducibles se rompen sin avisar cuando cambias una dependencia; y al refactorizar código de notebook a src/ querés saber que el comportamiento sigue siendo el mismo. Tests bien escritos son la red que te deja moverte rápido sin romper lo que ya funciona.

Estructura mínima: archivos test_.py dentro de tests/, funciones test_() adentro, y assert para verificar. pytest descubre todo automáticamente.

Concepto	Para qué sirve
assert	Verificación básica: assert resultado == e
pytest.fixture	Datos o objetos reutilizables entre tests
pytest.mark.parametrize	Corre el mismo test con varios inputs dist
pytest.raises	Verifica que una función levante la excepc
conftest.py	Fixtures compartidas entre varios archivos
--cov	Reporta cobertura de código (requiere pyte

Mini-ejemplo. Función en src/mi_proyecto/features.py:

```
def normalize(x):
    return (x - x.mean()) / x.std()
```

Test en tests/test_features.py:

```
import numpy as np
import pandas as pd
from mi_proyecto.features import normalize

def test_normalize_media_cero():
    s = pd.Series([1.0, 2.0, 3.0, 4.0, 5.0])
    result = normalize(s)
    assert np.isclose(result.mean(), 0.0)
    assert np.isclose(result.std(), 1.0)
```

Correr los tests: pytest -v para ver todo lo que corre, pytest --cov=src tests/ para incluir cobertura.

¿Qué testear en DS? Funciones puras de transformación (limpieza, encoding, feature engineering) sí — son determinísticas y rápidas. Modelos entrenados con aleatoriedad no se testean directamente sobre métricas exactas; o fijás random_state/seeds y verificás contra un valor congelado, o usás snapshots tolerantes (pytest.approx). Lo que importa es testear la lógica que escribiste, no reinventar scikit-learn.

Definiciones y características

Cookiecutter

: Generador de proyectos a partir de plantillas. Tomas una plantilla (URL de un repo), respondes 3-5 preguntas y obtienes un proyecto con estructura pre-armada. Característica: idempotente — la plantilla no sabe ni le importa el contenido futuro del proyecto.

CCDS (cookiecutter-data-science)

: Plantilla específica para proyectos de DS, v2 (2023+). Separa data/raw, data/interim, data/processed, src/, notebooks/, reports/, docs/. Es convención, no dogma.

Editable install (pip install -e .)

: Instala el paquete pero apuntando al código fuente — los cambios se reflejan sin reinstalar. Habilita from mi_proyecto.features import x desde notebooks dentro del repo.

pyproject.toml

: Estándar moderno (PEP 621) para metadata + dependencias + config de tools (ruff, mypy, pytest). Reemplaza setup.py + setup.cfg + requirements.txt suelto + configs sueltas.

Makefile

: Archivo con "recetas" nombradas (make data, make train). Escrito en tabs (no espacios), las dependencias se declaran arriba (target: dep1 dep2). En proyectos DS funciona como interfaz humana a comandos típicos.

Dataset / recursos

Para el ejercicio principal, descarga el dataset de Palmer Penguins (https://github.com/allisonhorst/palmerpenguins) — pequeño (~13 KB), público, ideal para que data/raw/penguins.csv ocupe poco y se pueda commitear sin mala práctica.

Ejercicios

1. Genera un proyecto CCDS. pipx run cookiecutter https://github.com/drivendataorg/cookiecutter-data-science con nombre ds-lab-004. Explora la estructura.

2. Mueve código a src/. Toma una función de un notebook tuyo previo y muévela a src/<proyecto>/features.py. Importa desde el notebook con from <proyecto>.features import ….

3. Convierte requirements.txt a pyproject.toml (sección [project.dependencies]).

4. Refactoriza un notebook caótico. Toma uno con bloques copy-paste y extrae 2 funciones a src/.

5. Lista 5 olores en un repo público que conozcas y propón cómo arreglarlos.

Homework verificable

Un repo público con estructura CCDS, data/raw/ con un dataset pequeño, al menos 1 notebook que importa funciones desde src/, pyproject.toml con deps, Makefile con make setup y make data.

Criterio de aceptación: Un compañero clona, corre make setup && make data y obtiene la misma estructura de datos procesados que tú reportaste. README explica el flujo.

Errores comunes

Síntoma / mensaje	Causa y cómo arreglar
ModuleNotFoundError: No module named 'mi_p	El paquete no está instalado en el venv de
CSV en data/raw/ apareció en git status y	El .gitignore no cubre data/raw/ o no se a
Tengo 3 notebooks con la misma función lim	Copy-paste. Fix: extrae a src/mi_proyecto/
El compañero clonó el repo y make data fal	Make no está instalado en Windows por defa
Edité pyproject.toml y los imports siguen	Tras cambios en metadata o entry-points de
pytest corre pero dice collected 0 items	Los archivos o funciones no respetan la co

Preguntas frecuentes

¿Realmente necesito una plantilla? ¿No puedo improvisar?

Puedes, pero perderás el 80% de la ganancia: el compañero que ya conoce CCDS sabe dónde buscar; sin plantilla, cada proyecto es una caja de sorpresas.

¿data/raw/ o data/01_raw/?

CCDS v2 usa data/raw/, data/interim/, data/processed/, data/external/. La numeración 01_/02_/03_ la verás en Kedro y en algunas variantes — ambas son válidas, sigue una sola convención por proyecto.

¿requirements.txt o pyproject.toml?

pyproject.toml para declarar las deps de tu paquete. requirements.txt (generado con pip freeze o uv pip compile) es el lockfile con versiones exactas para reproducir. No están en conflicto; conviven.

¿Y si el proyecto es solo un notebook exploratorio?

No fuerces CCDS. Carpeta con notebook.ipynb, data.csv y README.md está bien. La estructura completa aporta cuando el proyecto vive >3 meses o tiene >1 persona.

¿Por qué los notebooks tienen prefijo numérico como 0.01-jvp-eda.ipynb?

Convención CCDS: <fase>.<orden>-<iniciales>-<tema>. Fase 0=exploración, 1=features, 2=modelos, 3=reportes. Iniciales del autor evitan conflictos cuando varias personas crean notebooks.

¿Hace falta testear notebooks?

Los notebooks no se testean directamente — son scratchpads exploratorios y cambian todo el tiempo. Lo que sí se testea es el código que migrás del notebook a src/: en el momento que una función deja de ser exploración y pasa a src/mi_proyecto/, le escribís su test_* correspondiente. Regla práctica: si la función vive en src/, tiene test; si vive en una celda, no.

Referencias

- cookiecutter-data-science v2 docs
- Sculley et al., Hidden Technical Debt in ML Systems (NeurIPS 2015).
- Palmer Penguins dataset
- pytest docs

Siguiente clase

Clase 005 — VS Code / Cursor para Python y Jupyter

Apéndice: notebook (primer bloque)

Regla: nunca modifiques un archivo en data/raw/. Todo procesamiento escribe a data/interim/ o data/processed/. Por qué: - Si tu pipeline rompe, puedes regenerar todo desde el origen.

```
# Demo: estructura típica creada con Path (simulación, no genera nada en disco)
from pathlib import Path

estructura = [
    'mi-proyecto/data/raw/',
    'mi-proyecto/data/interim/',
    'mi-proyecto/data/processed/',
    'mi-proyecto/notebooks/0.01-eda.ipynb',
    'mi-proyecto/src/mi_proyecto/__init__.py',
    'mi-proyecto/src/mi_proyecto/features.py',
    'mi-proyecto/pyproject.toml',
    'mi-proyecto/Makefile',
    'mi-proyecto/README.md',
```

```
]
for p in estructura:
    icon = '' if p.endswith('/') else ''
    print(f'{icon} {p}')
```

Archivos complementarios

- notebook.ipynb