Ingeniería de IA I

Fundamentos de Ciencia de Datos: Terminal, Git, Python y Docker

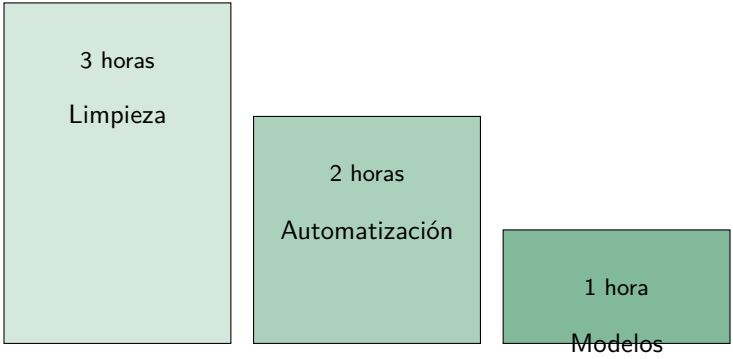
AgroFuture AI Training

Departamento de Ingeniería
Valle del Cauca, Colombia

Enero 2026

Índice

Realidad del Científico de Datos



Por cada hora entrenando modelos:

- 3 horas de limpieza y validación de datos
- 2 horas de automatización y scripting
- 1 hora de documentación y versionado

Principios Fundamentales

Reproducibilidad

Un modelo que no puede reproducirse no es ciencia: es magia.

¿Qué aprenderemos?

1. **Terminal Linux**: navegar, procesar datos
2. **Git**: versionado y control de cambios
3. **Python**: automatización y análisis
4. **Docker**: ambientes reproducibles

Navegación Básica

- `pwd`: saber en qué carpeta estás actualmente
- `ls -lh`: listar archivos con tamaños legibles
- `cd carpeta`: cambiar a una carpeta
- `mkdir -p ruta`: crear carpetas (y subcarpetas)

Objetivo

Aprender a orientarse en el sistema de archivos de un proyecto de datos.

Navegación Básica: ejemplo

Sesión típica

```
cd ~/proyecto_ia
mkdir -p data/raw data/processed
ls -lh data/
```

Esto crea carpetas para datos brutos y procesados, y verifica que existan.

Manipulación de Archivos

- `cp archivo.txt copia.txt`: copiar (usa `-i` para preguntar)
- `mv viejo.txt nuevo.txt`: mover o renombrar
- `rm -i archivo.txt`: eliminar con confirmación
- `du -sh *`: ver tamaño de carpetas
- `wc -l datos.csv`: contar filas

Manipulación de Archivos: ejemplo

Organizar proyecto

```
cd ~/proyecto_ia
mkdir -p scripts reports
cp ~/sensores.csv data/raw/sensores.csv
mv README_tmp.md README.md
```

Las operaciones básicas permiten organizar un proyecto profesional.

Pipelines con Pipes

El operador | encadena comandos: salida de uno = entrada del siguiente.

- `cut`: extrae columnas
- `sort`: ordena filas
- `uniq`: elimina duplicados

Ventaja: sin crear archivos intermedios, sin abrir editores gráficos.

Pipelines con Pipes: ejemplo

Análisis en una línea

```
# Contar registros únicos por sensor
cut -d',' -f1 sensores.csv | sort | uniq -c

# Ver las 5 temperaturas más altas
cut -d',' -f3 sensores.csv | sort -nr | head -5
```

Procesamiento con awk

`awk` es un mini-lenguaje para cálculos y manipulación en datos tabulares.

- Extrae columnas específicas
- Filtra filas por condiciones
- Calcula sumas, promedios, máximos
- Genera reportes directamente

Procesamiento con awk: ejemplos

Cálculos estadísticos

```
# Temperatura promedio
awk -F',' '{s+=$3} END {print s/NR}' sensores.csv

# Contar registros con humedad < 30
awk -F',' '$4 < 30 {count++} END {print count}' sensores.csv

# Guardar filas con humedad > 80
awk -F',' '$4 > 80 {print}' sensores.csv > alertas.csv
```

¿Por Qué Git?

Sin Git:

- `modelo_v1.pth`
- `modelo_v2_final.pth`
- `modelo_v2_final2.pth`
- ¿Quién cambió qué?
- ¿Cuándo?

Con Git:

- Una versión actual
- Historial completo
- Quién hizo qué
- Cuándo
- Por qué (mensaje)

Flujo Básico de Git

1. `git init`: inicializar repositorio
2. `git status`: ver qué cambió
3. `git add .`: seleccionar cambios
4. `git commit -m "mensaje"`: guardar con descripción
5. `git log --oneline`: ver historial resumido

Flujo Básico de Git: ejemplo

Primera sesión completa

```
cd proyecto_ia
git init
git add .
git commit -m "Inicial: estructura del proyecto"
git log --oneline
```

Así comienza cualquier proyecto profesional de ciencia de datos.

Buenas Prácticas: .gitignore

¿Qué NO debe versionarse?

- `data/raw/`: datos brutos sin procesar
- `*.pth, *.h5`: modelos entrenados (muy grandes)
- `venv/`: entornos virtuales
- `.ipynb_checkpoints/`: archivos temporales
- `__pycache__/`: cache de Python

Buenas Prácticas: .gitignore (crear)

Crear archivo .gitignore

```
cat > .gitignore << EOF
venv/
data/raw/
*.pth
.ipynb_checkpoints/
__pycache__/
EOF

git add .gitignore
git commit -m "Configura .gitignore"
```

Mensajes de Commit Claros

Evitar

- çambios"
- .ªrreglado"
- "v2"

Hacer

- .Añade validación de valores faltantes"
- Çorrige filtro de humedad en CSV"
- Ïmplementa pipeline de limpieza automática"

Entornos Virtuales: Problema

- Proyecto A necesita `pandas 1.0`
- Proyecto B necesita `pandas 2.0`
- `pip install pandas==2.0` rompe Proyecto A

Solución

Cada proyecto su propio entorno aislado.

Entornos Virtuales: crear

Crear un entorno

```
python -m venv ia_env
source ia_env/bin/activate
pip install pandas numpy scikit-learn
pip freeze > requirements.txt
```

`requirements.txt` guarda las versiones exactas instaladas.

Entornos Virtuales: reproducir

Recrear el mismo entorno en otra máquina

```
python -m venv ia_env
source ia_env/bin/activate
pip install -r requirements.txt
```

Garantiza que otros colaboradores y servidores tengan las mismas versiones.

Pipeline Shell + Python

Combinar lo mejor de ambos mundos

```
#!/bin/bash
source ia_env/bin/activate

# Procesar con shell
cut -d',' -f1,3,4 data/raw/sensores.csv > temp.csv

# Analizar con Python
python - << EOF
import pandas as pd
df = pd.read_csv('temp.csv')
print(df.describe())
EOF
```

¿Por Qué Docker?

Mi máquina:

- Python 3.10
- pandas 2.0
- Funciona

En el servidor:

- Python 3.8
- pandas 1.5
- ¡Error!

Docker: ."Empaqueta tu ambiente completo"

Conceptos Básicos de Docker

Imagen

Plantilla inmutable: `FROM python:3.10` + librerías + código

Contenedor

Instancia en ejecución de una imagen

Dockerfile

Archivo que describe cómo construir la imagen

Registry

Lugar donde publicas imágenes (p.ej. Docker Hub)

Dockerfile para Ciencia de Datos

Ejemplo mínimo

```
FROM python:3.10-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
EXPOSE 8888
CMD ["jupyter", "lab", "--ip=0.0.0.0"]
```

Partes de un Dockerfile

- `FROM`: imagen base oficial (p.ej. `python:3.10-slim`)
- `WORKDIR`: directorio de trabajo dentro del contenedor
- `COPY`: copiar archivos locales
- `RUN`: instalar dependencias (pip install, apt-get, etc.)
- `EXPOSE`: puertos que expone (documentación)
- `CMD`: comando por defecto al iniciar

Construcción y Ejecución

Construir la imagen

```
docker build -t proyecto_ia:latest .
```

Ejecutar un contenedor

```
docker run --rm \
  -v $(pwd)/data:/app/data \
  -v $(pwd)/reports:/app/reports \
  proyecto_ia:latest
```

Opciones de docker run

- `-v carpeta_local:/app/carpeta`: montar carpeta local dentro del contenedor
- `--rm`: borrar contenedor al terminar (limpieza)
- `-p 8888:8888`: exponer puerto (para Jupyter)
- `-it`: modo interactivo (para shells)

Los datos en tu máquina siempre están seguros; el contenedor solo ve lo que le permitas.

Docker Compose: Múltiples Servicios

docker-compose.yml

```yaml
version: "3.8"
services:
  db:
    image: postgres:13
  notebook:
    build: .
    depends_on:
      - db
    ports:
      - "8888:8888"
```

```
docker compose up    # Levanta ambos
docker compose down  # Los apaga
```

Carpetas Recomendadas

- `data/raw/`: datos sin procesar
- `data/processed/`: datos limpios
- `scripts/`: código reutilizable
- `notebooks/`: exploración Jupyter
- `models/`: modelos entrenados
- `reports/`: figuras y tablas
- `README.md`: descripción
- `requirements.txt`: dependencias

Regla fundamental

Si no está versionado en Git, no existe.

Crear Estructura Completa

Una sola orden

```
mkdir -p proyecto_ia/{data/raw,data/processed,\
scripts,notebooks,models,reports}

cd proyecto_ia
git init
touch README.md
git add .
git commit -m "Inicial: estructura"
```

De Cero a Análisis

1. Crear estructura con `mkdir -p`
2. `git init` + crear `.gitignore`
3. `python -m venv ia_env`
4. Descargar datos a `data/raw/`
5. Scripts de limpieza en `scripts/`
6. Análisis en notebooks
7. Reportes en `reports/`
8. Commits frecuentes con mensajes claros
9. Crear `Dockerfile` para reproducibilidad

Checklist Profesional

- ✓ Código versionado en Git
- ✓ Datos en carpeta `raw/`, nunca modificar
- ✓ Scripts reproducibles, no manuales
- ✓ Entorno virtual con `requirements.txt`
- ✓ README explicando el proyecto
- ✓ Dockerfile para ejecutar en cualquier máquina
- ✓ Datos sensibles en `.gitignore`
- ✓ Commits pequeños y frecuentes

Proyecto Completo: Monitoreo de Sensores

Construye un proyecto que:

1. Descargue datos de sensores (archivo CSV)
2. Valide datos: cuente filas, busque faltantes
3. Limpie datos automáticamente
4. Genere gráficos y tablas
5. Guarde resultados en `reports/`
6. Todo versionado en Git
7. Todo reproducible en Docker

Requisito: Idempotencia

Ejecutar 10 veces debe dar el mismo resultado sin errores.

Componentes del Reto

Script de descarga

`scripts/descargar_datos.sh`: obtiene CSV de fuente confiable

Script de validación

`scripts/validar_datos.py`: estadísticas y calidad

Script de limpieza

`scripts/limpiar_datos.py`: elimina errores, guarda en `processed/`

Dockerfile

Ejecuta todo automáticamente en un contenedor

Nivel Intermedio

Herramientas complementarias:

- MLflow: experiment tracking y reproducibilidad
- GitHub Actions: CI/CD automático
- DVC: versionado de datos grandes
- Airflow: orquestación de pipelines

Nivel Avanzado

Productividad y escalabilidad:

- FastAPI: servir modelos como APIs
- Kubernetes: orquestación de contenedores en producción
- Prometheus + Grafana: monitoreo
- GitLab CI/CD o Jenkins: pipelines complejos

Recordar

La reproducibilidad es lo más importante.

Un código sin documentación es deuda técnica.
Un modelo sin versión es imposible de auditar.
Datos sin respaldo desaparecen.

Las herramientas que aprendiste garantizan que tu trabajo sea sólido y duradero.

¿Preguntas?

Documentación oficial:

- Bash: `https://www.gnu.org/software/bash/manual/`
- Git: `https://git-scm.com/doc`
- Docker: `https://docs.docker.com/`
- Python venv: `https://docs.python.org/3/library/venv.html`

¡Muchas gracias!