Clase 2: Conceptos Básicos de Git en Acción para Científicos de Datos

Objetivos de la Clase

Al finalizar esta clase, serás capaz de:

- Crear y **clonar repositorios** de Git.
- Comprender a fondo el **ciclo de vida de los archivos** en Git y cómo mover tus scripts y notebooks entre los estados (`Untracked`, `Modified`, `Staged`, `Committed`).
- Realizar y gestionar **commits** de manera efectiva para registrar tus cambios y experimentos de datos.
- Utilizar comandos básicos como `git add`, `git commit`, `git status`, `git log`, y `git diff`.
- Entender la importancia de la **visualización de cambios** en tus proyectos de ciencia de datos.

1. Creación y Clonación de Repositorios

Un repositorio Git es esencialmente una base de datos de tu proyecto donde se almacena todo su historial de cambios, desde la primera línea de código de un script de preprocesamiento hasta la última versión de tu modelo predictivo. Puedes crear uno desde cero o traer uno ya existente.

1.1. Inicializar un Nuevo Repositorio Local (`git init`)

Como vimos brevemente en la Clase 1, `git init` es el comando que convierte cualquier directorio en un repositorio Git. Este comando es fundamental cuando inicias un nuevo proyecto de ciencia de datos desde cero en tu máquina.

Paso a Paso:

1. Navega a la carpeta donde quieres iniciar tu proyecto de datos (o crea una nueva).

```
Unset
# Si quieres una nueva carpeta para tu proyecto de análisis de datos

mkdir mi-proyecto-ia

cd mi-proyecto-ia
```

```
# Si ya estás en tu carpeta de proyecto de datos existente

# cd /ruta/a/tu/analisis_de_datos_existente
```

2. Inicializa el repositorio Git:

```
Unset
git init
```

3. Verás un mensaje como: Initialized empty Git repository in /ruta/a/mi-proyecto-ia/.git/.
4. Esto crea la subcarpeta oculta .git, que es el corazón de tu repositorio local. Contendrá todo el historial de tus scripts, modelos y resultados.

Concepto Clave:

- `git init` solo crea un **repositorio local**. Los cambios que hagas solo estarán en tu máquina. Para colaborar o tener una copia de seguridad remota, necesitarás un repositorio remoto (que veremos en la Clase 5).
- Por defecto, Git crea una rama principal llamada `master` o `main`. Puedes configurar el nombre predeterminado con `git config --global init.defaultBranch main` si deseas que sea `main` siempre, lo cual es la convención moderna.

1.2. Clonar un Repositorio Existente (`git clone`)

Cuando quieres empezar a trabajar en un proyecto de ciencia de datos que ya existe (por ejemplo, el repositorio de un colega, un proyecto de código abierto con datasets, o uno que tienes alojado en GitHub/GitLab), lo "clonas". Clonar significa descargar una copia completa del repositorio, incluyendo todo su historial.

Paso a Paso:

1. Obtén la URL del repositorio que deseas clonar. Esta puede ser una ruta local a otro repositorio en tu máquina o una URL remota (HTTPS o SSH).
2. Abre tu terminal en la ubicación donde quieres que se cree la carpeta del proyecto.
3. Ejecuta el comando `git clone` seguido de la URL:

```
Unset
# Clonar desde una URL HTTPS (común si no tienes SSH configurado)

git clone https://github.com/tu-organizacion/analisis-fraude.git

# Clonar desde una URL SSH (recomendado por seguridad y comodidad)

git clone git@github.com:tu-organizacion/analisis-fraude.git
```

4. Git descargará una copia completa del repositorio (todos los scripts de Python, notebooks Jupyter, ramas e historial) en una nueva carpeta con el mismo nombre que el repositorio (ej. `analisis-fraude`).
   Opcional: Si quieres que la carpeta local tenga un nombre diferente, puedes especificarlo:

```
Unset
git clone https://github.com/tu-organizacion/analisis-fraude.git
mi-analisis-local
```

Errores Comunes y Soluciones:

- `fatal: repository 'URL' not found`: La URL es incorrecta, el repositorio no existe, o no tienes permisos de acceso. Verifica la URL y tus permisos en la plataforma remota.
- `Permission denied (publickey)` **(al usar SSH)**: Tu clave SSH no está configurada correctamente en tu sistema o en la plataforma remota (GitHub/GitLab). Esto se cubrirá en detalle en la Clase 6.

2. Concepto de Ramas (Branches)

Las ramas son una de las características más potentes y fundamentales de Git. Permiten el **desarrollo paralelo y seguro**, crucial en ciencia de datos para la experimentación.

Una **rama** es, en esencia, una línea independiente de desarrollo dentro de tu repositorio. Cuando creas una rama, Git crea una "copia" del estado actual de tu código y datos, permitiéndote trabajar en ella sin afectar la rama principal u otras ramas.

Image of Un poco de teoría - Qué es una rama

¿Para qué se utilizan las ramas en ciencia de datos?

- **Aislamiento de Experimentos:** Puedes desarrollar un nuevo algoritmo de clasificación, probar una técnica de reducción de dimensionalidad, o refactorizar tu script de preprocesamiento en una rama separada, sin introducir inestabilidad en tu modelo principal o script de producción.
- **Colaboración en Módulos:** Varios científicos de datos pueden trabajar en diferentes módulos de un pipeline (uno en preprocesamiento, otro en entrenamiento, otro en evaluación) en paralelo, cada uno en su propia rama.
- **Reproducción de Bugs/Anomalías:** Si encuentras un error o una anomalía en un análisis anterior, puedes crear una rama específica para investigar y corregirlo, sin afectar tu trabajo actual.
- **Versiones de Modelos/Análisis:** Puedes usar ramas para mantener diferentes versiones de modelos o análisis exploratorios antes de decidir cuál integrar.

Concepto Clave:

- La rama principal de un repositorio suele llamarse `master` o `main`. Esta es la versión "estable" o "de producción" de tu código y modelos.
- Las ramas pueden **bifurcarse** (separarse del flujo principal) y luego **fusionarse** (integrarse de nuevo) con otras ramas.

3. El Concepto de Commit: La "Fotografía" de Tu Proyecto de Datos

El commit es la unidad fundamental de registro de cambios en Git.

Piensa en un commit como una "fotografía" instantánea de tu proyecto en un momento dado. Cada commit captura el estado de todos los archivos rastreados (scripts, notebooks, configuraciones de modelo) en tu repositorio en el momento en que lo creaste.

Image of Trabajando con Git de forma local - Qué es un commit

Importancia de los Commits para Científicos de Datos:

- **Registro del Historial de Experimentos:** Cada `commit` se convierte en un punto en el historial de tu proyecto al que puedes regresar en cualquier momento. Esto es útil para comparar versiones de modelos, scripts de preprocesamiento o análisis.
- **Trazabilidad y Auditoría:** Cada `commit` incluye metadatos importantes:
    - Un identificador único (un hash SHA-1, ej., `e137e9b`).
    - El autor del cambio (tu nombre y email).
    - La fecha y hora del cambio.

    - Un **mensaje de commit** que describe lo que se modificó (ej. "Ajuste de hiperparámetros para el modelo X", "Implementación de k-fold cross-validation") y por qué.
- **Unidad Atómica del Trabajo:** Un buen `commit` representa un cambio lógico y autocontenido. No es solo un archivo modificado, sino un conjunto de cambios (ej. la actualización de un script de limpieza Y la adición de un test para esa limpieza) que juntos tienen un significado y propósito.

Analogía: Si el área de staging es la preparación de tus instrumentos y datos antes de un registro, el `commit` es el momento en que tomas los datos y los guardas como un registro de experimento completo, y el repositorio es tu cuaderno de laboratorio digital con todas tus entradas.

4. Ciclo de Vida de los Archivos en Git: Profundizando

Como aprendimos en la Clase 1, los archivos en Git pasan por diferentes estados. Ahora, veamos cómo interactúan con los comandos para moverse entre ellos, con ejemplos de ciencia de datos.

Image of Un poco de teoría - Los tres estados en Git

4.1. Estado `Untracked` (Sin Rastrear)

Cuando creas un nuevo archivo en tu **Directorio de Trabajo**, Git no lo rastrea automáticamente. Esto es útil para archivos temporales o datasets muy grandes que no quieres versionar.

- **Comando para identificar:** `git status`

Ejemplo:

```
Unset

# Después de crear un nuevo script o notebook

touch new_feature_script.py

touch exploratory_analysis.ipynb # Notebook Jupyter

git status

# Salida:
```

```
# Untracked files:

#   (use "git add <file>..." to include in what will be committed)

#       new_feature_script.py

#       exploratory_analysis.ipynb
```

4.2. Estado `Modified` (Modificado)

Si modificas un script o archivo que Git ya está rastreando (es decir, que ya ha sido parte de un `commit` anterior), ese archivo pasa al estado `Modified`.

- **Comando para identificar:** `git status`

Ejemplo:

Unset

```
# Después de modificar un script de preprocesamiento existente

# (Asumiendo que preprocess_data.py ya fue commiteado)

echo "import pandas as pd\n\ndef clean_data(df): return df.dropna()"
> preprocess_data.py # Modifica el archivo

git status

# Salida:

# Changes not staged for commit:

#   (use "git add <file>..." to update what will be committed)

#   (use "git restore <file>..." to discard changes in working
directory)

#       modified:   preprocess_data.py
```

4.3. Estado `Staged` (Preparado)

Para incluir los cambios de un archivo (`Untracked` o `Modified`) en el próximo `commit`, debes moverlo al **Área de Staging**. Esto se hace con el comando `git add`. Piensa en esto como "preparar" los resultados o las nuevas funcionalidades para el registro oficial.

- **Comando para mover a `Staged`:** `git add <ruta_archivo>`

Ejemplo:

```
Unset

# Mover preprocess_data.py al área de staging

git add preprocess_data.py

git status

# Salida:

# Changes to be committed:

#   modified:   preprocess_data.py
```

Variantes de `git add`:

- `git add .`: Añade todos los archivos modificados y no rastreados en el directorio actual (y subdirectorios) al área de staging.
- `git add -A` o `git add --all`: Añade todos los archivos modificados, eliminados y no rastreados en todo el repositorio al área de staging.

4.4. Estado `Committed` (Confirmado)

Cuando los cambios de un archivo están en el Área de Staging, puedes guardarlos permanentemente en el **Repositorio Local** creando un `commit`. Esta es tu "entrada de laboratorio" final.

- **Comando para mover a `Committed`:** `git commit`

Ejemplo:

Unset

```
# Con cambios en staging, creamos el commit

git commit -m "feat: Implementación inicial de limpieza de datos en preprocess_data.py"

# Salida:

# [main e137e9b] feat: Implementación inicial de limpieza de datos en preprocess_data.py

#  1 file changed, 2 insertions(+)

#  create mode 100644 preprocess_data.py

git status

# Salida:

# nothing to commit, working tree clean
```

Ahora el script `preprocess_data.py` está `Committed`.

5. Comandos Básicos en Acción: Paso a Paso

Vamos a poner en práctica el ciclo de vida de los archivos con los comandos fundamentales, utilizando nuestro contexto de ciencia de datos.

5.1. `git status`: Conoce el Estado de Tu Proyecto de Datos

`git status` es el comando más importante para entender qué está pasando en tu repositorio. Te muestra los scripts y notebooks que han sido modificados, los que están en el área de staging, los nuevos archivos sin rastrear, y si tu rama está al día con el repositorio remoto.

Sintaxis:

Unset

```
git status
```

Sintaxis Simplificada:

Si el resultado de git status te parece demasiado verboso, puedes usar el parámetro -s (o --short) para una salida más concisa:

```
Unset

git status -s
```

Leyenda para git status -s:

La salida corta usa dos columnas: la primera indica el estado del staging area y la segunda el working directory.

- `??`: Untracked (sin rastrear) - un nuevo archivo de dataset
- `A `: Staged (añadido) - un script que has preparado para el commit
- `M `: Staged (modificado) - un notebook que has modificado y preparado
- `D`: Staged (eliminado) - un archivo de configuración que has eliminado y preparado
- `M`: Modified (modificado en el directorio de trabajo) - un script de entrenamiento que has editado pero no preparado
- `D`: Deleted (eliminado en el directorio de trabajo) - un archivo que has borrado pero no preparado
- `R `: Renamed (renombrado)
- `C `: Copied (copiado)
- `U `: Unmerged (conflictos)
- `! `: Ignored (ignorado)

Ejemplo:

```
Unset

# Después de crear un nuevo notebook y modificar un script existente
```

```
touch model_evaluation.ipynb

echo "print('Modelo evaluado!')" >> predict_script.py #
predict_script.py ya estaba commiteado

git status -s

# Salida:

# ?? model_evaluation.ipynb

#  M predict_script.py
```

Esto significa: `model_evaluation.ipynb` es un nuevo notebook y sin rastrear. `predict_script.py` ha sido modificado en el directorio de trabajo.

5.2. `git add`: Prepara Tus Cambios para el Registro

Usa `git add` para mover los cambios de tu **Directorio de Trabajo** al **Área de Staging**, indicando a Git qué quieres incluir en el próximo `commit`. Es como seleccionar qué resultados o versiones de scripts vas a documentar.

Sintaxis:

```
Unset


git add <ruta_del_archivo>      # Para un archivo específico (ej.
model.py)

git add <directorio>/           # Para todos los cambios en un
directorio (ej. data_processing/)

git add .                       # Para todos los cambios en el
directorio actual (común)

git add -A # o --all            # Para todos los cambios en todo el
repositorio (incluye eliminaciones)
```

Ejemplo:

```
Unset

# Creamos un archivo de requisitos para el proyecto

touch requirements.txt

git status # requirements.txt aparece como Untracked


# Añadimos requirements.txt al área de staging

git add requirements.txt

git status

# Salida:

# Changes to be committed:

#   new file:   requirements.txt
```

5.3. `git commit`**: Guarda Tus Cambios Como un Registro Oficial**

Una vez que tienes los cambios en el **Área de Staging**, `git commit` los guarda como una nueva "fotografía" en el **Repositorio Local**. Cada commit debe ser un punto significativo y coherente en tu proyecto.

Sintaxis (con mensaje corto):

```
Unset

git commit -m "feat: Añadir script para generación de features"
```

El mensaje debe ser conciso y describir el propósito de los cambios.

Sintaxis (con editor para mensaje largo):

```
Unset

git commit
```

Esto abrirá el editor de texto que configuraste en la Clase 1. La primera línea será el título del commit, y las líneas posteriores el cuerpo del mensaje. Las líneas que empiezan con `#` son comentarios y no se guardan.

Saltarse el Área de Staging para Archivos Rastreados (-a o --all):

Puedes combinar git add y git commit para archivos que Git ya está rastreando (Modified pero no Untracked).

```
Unset

# Modificamos un script de evaluación ya rastreado (no nuevo)

echo "from sklearn.metrics import accuracy_score\nprint('Eval OK')"
>> evaluate_model.py

git status # evaluate_model.py aparece como Modified



# Commit directo sin git add previo para este archivo

git commit -am "refactor: Mejorar métricas de evaluación en
evaluate_model.py"
```

Importante: La opción `-a` solo funciona para archivos que ya han sido **rastreados previamente por Git**. Si es un archivo completamente nuevo (`Untracked`), primero debes usar `git add` para que Git comience a rastrearlo.

Errores Comunes y Soluciones:

- `fatal: no changes added to commit (use "git add" and/or "git commit -a")`: Significa que no hay cambios en el Área de Staging, ni archivos modificados y rastreados para usar `-a`. Debes usar `git add` primero.

5.4. `git log`: Explora el Historial de tus Experimentos

`git log` te permite ver el historial de commits de tu repositorio. Cada entrada muestra el hash del commit, autor, fecha y el mensaje. Esto es como revisar tu cuaderno de laboratorio cronológicamente.

Sintaxis:

```
Unset

git log
```

Sintaxis Útiles para Visualizar el Historial:

- `git log --oneline`: Muestra cada commit en una sola línea (hash abreviado y mensaje). Ideal para una visión rápida.
- `git log --graph`: Muestra el historial en un formato de grafo ASCII, útil para visualizar cómo se han creado y fusionado las ramas de experimentos.
- `git log --oneline --graph --all`: Una combinación poderosa para ver el historial de todas las ramas de forma concisa y gráfica.

Ejemplo:

```
Unset

git log --oneline

# Salida:

# a1b2c3d (HEAD -> main) feat: Añadir función para visualización de clusters

# e4f5g6h fix: Corregir error en script de preprocesamiento

# i7j8k9l Initial project setup
```

5.5. `git diff`: Visualiza las Diferencias en Tu Código y Datos

`git diff` es esencial para ver las diferencias entre diferentes versiones de tus scripts o notebooks. Te ayuda a entender exactamente qué ha cambiado.

Sintaxis y Usos Comunes:

- `git diff`: Muestra los cambios en tu **Directorio de Trabajo** que aún no han sido añadidos al **Área de Staging** (archivos `Modified`).
- `git diff --staged` (o `git diff --cached`): Muestra los cambios que están en el **Área de Staging** y que aún no han sido commiteados.
- `git diff <commit1> <commit2>`: Compara dos commits específicos (útil para ver qué cambió entre dos versiones de un modelo).
- `git diff HEAD`: Compara el **Directorio de Trabajo** con el último `commit` (lo que has hecho desde la última "fotografía").

Ejemplo:

Unset

```
# Editamos un script de análisis

# Contenido original: print("Análisis inicial")

# Nuevo contenido en analyze_data.py:

# import numpy as np

# print("Análisis avanzado con numpy")

git diff analyze_data.py

# Salida (ejemplo simplificado):

# --- a/analyze_data.py

# +++ b/analyze_data.py

# @@ -1 +1,2 @@

# -print("Análisis inicial")

# +import numpy as np

# +print("Análisis avanzado con numpy")
```

6. Deshaciendo Cambios (Antes de Commitear)

A veces, cometes un error al escribir un script o simplemente cambias de opinión sobre un enfoque antes de hacer un `commit`. Git te ofrece redes de seguridad para deshacer estos cambios sin afectar tu historial.

6.1. Deshacer Cambios en el Directorio de Trabajo (`git restore`)

Si has modificado un script o notebook existente y quieres volver a su versión anterior (la última que fue commiteada o stashead), usa `git restore`.

Sintaxis:

```
Unset


git restore <nombre_archivo.py>    # Para un script específico

git restore <nombre_notebook.ipynb> # Para un notebook

git restore .                       # Para descartar todos los
cambios en el directorio de trabajo (¡Cuidado!)
```

Importante: `git restore` solo funciona para archivos que Git ya está rastreando (`Modified`). Si el archivo es `Untracked` (nuevo y no añadido a Git), `git restore` te dará un error porque no tiene una versión anterior a la cual restaurar.

Ejemplo:

```
Unset


# Modificamos script_para_pruebas.py

# Contenido original: def test_func(): pass

echo "def test_func(): return 1 / 0 # Error a propósito" >
script_para_pruebas.py
```

```
git status # script_para_pruebas.py aparece como Modified


# Deshacemos los cambios

git restore script_para_pruebas.py

git status # script_para_pruebas.py vuelve a su estado original
```

6.2. Sacar Archivos del Área de Staging (`git reset <archivo>`)

Si has usado `git add` para preparar los cambios de un script, pero luego te arrepientes de incluir esos cambios en el próximo `commit`, puedes sacarlos del área de staging.

Sintaxis:

```
Unset

git reset <nombre_archivo.py>
```

Esto mueve el archivo de `Staged` a `Modified`. Los cambios no se pierden, solo dejan de estar "preparados".

Ejemplo:

```
Unset

# Modificamos config.ini y lo añadimos al staging

echo "[MODEL]\nlearning_rate = 0.001" > config.ini

git add config.ini

git status # config.ini aparece como Changes to be committed
```

```
# Lo sacamos del staging

git reset config.ini

git status # config.ini ahora aparece como Changes not staged for commit
```

6.3. Eliminar Archivos No Rastreados (`git clean`)

Si has creado archivos nuevos (`Untracked`), como datasets temporales, resultados intermedios o scripts de prueba rápidos, y decides que no los quieres en tu proyecto o que no deben ser versionados, puedes eliminarlos de forma segura.

Sintaxis y Opciones:

- `git clean -n` (o `--dry-run`): **Simula** la eliminación. Te muestra qué archivos se borrarían sin borrarlos realmente. **¡Siempre usa esto primero para evitar pérdidas accidentales!**
- `git clean -f` (o `--force`): **Fuerza** la eliminación de los archivos no rastreados.
- `git clean -d`: Borra también directorios vacíos que no están rastreados. Combínalo con `-f` (ej. `git clean -fd`).
- `git clean -i`: Modo interactivo. Te pregunta antes de borrar cada archivo/directorio.

Ejemplo:

```
Unset

# Creamos un archivo temporal de resultados

touch temp_results.csv

mkdir temp_outputs # y un directorio temporal

git status # Aparecen como Untracked


# Simular eliminación de archivos y directorios no rastreados
```

```
git clean -nfd

# Salida: Would remove temp_results.csv y el directorio
temp_outputs/



# Realizar eliminación

git clean -fd

git status # temp_results.csv y temp_outputs/ han desaparecido
```

7. Actividad Práctica (Laboratorio 2)

Es hora de aplicar todo lo aprendido. Sigue estos pasos en tu terminal, usando nuestro proyecto de análisis de datos.

Paso 1: Preparación del Entorno

1. Crea un nuevo directorio para esta clase y entra en él:

```
Unset
mkdir clase2-lab-ds

cd clase2-lab-ds
```

2. Inicializa un nuevo repositorio Git:

```
Unset
git init
```

3. Confirma con `git status`.

Paso 2: Creación de Archivos de Datos/Scripts y Estados `Untracked`

1. Crea tres archivos vacíos que representen componentes de un proyecto de datos:

```
Unset
touch data_ingestion.py model_training.py README.md
```

2. Verifica el estado actual de tu repositorio:

```
Unset
git status

# Observa que los tres archivos aparecen como Untracked.
```

Paso 3: Mover a `Staged` **(**`git add`**)**

1. Añade `README.md` al área de staging:

```
Unset
git add README.md
```

2. Verifica el estado:

```
Unset
git status

# Observa que README.md ahora está como Changes to be committed
(Staged),

# mientras data_ingestion.py y model_training.py siguen como
Untracked.
```

3. Añade `data_ingestion.py` y `model_training.py` al área de staging con un solo comando:

```
Unset
git add .
```

4. Verifica el estado de nuevo:

```
Unset
git status

# Ahora todos deberían estar en Changes to be committed.
```

Paso 4: Realizar el Primer Commit

1. Crea tu primer commit. Asegúrate de usar un mensaje claro y conciso:

```
Unset
git commit -m "Initial commit: Add data ingestion and model training
scripts"
```

2. Verifica el estado final del repositorio:

```
Unset
git status

# Deberías ver: nothing to commit, working tree clean.
```

Paso 5: Modificar Archivos y Ver Cambios (Modified **y** git diff**)**

1. Abre data_ingestion.py con tu editor de texto y añade contenido simulado de ingesta de datos:

```
Unset
# data_ingestion.py

import pandas as pd


def load_data(filepath):

    print(f"Loading data from {filepath}")

    df = pd.read_csv(filepath)
```

```
    return df


if __name__ == "__main__":
    # Simulación de carga de datos
    dummy_data = pd.DataFrame({'col1': [1,2], 'col2': [3,4]})
    dummy_data.to_csv('dummy_dataset.csv', index=False)
    data = load_data('dummy_dataset.csv')
    print(f"Data loaded with {len(data)} rows.")
```

2. Abre `model_training.py` y añade una simulación simple de entrenamiento:

```
Unset
# model_training.py
from sklearn.linear_model import LogisticRegression


def train_model(X, y):
    model = LogisticRegression()
    model.fit(X, y)
    print("Model trained successfully.")
    return model


if __name__ == "__main__":
    # Simulación de entrenamiento
```

```
    from sklearn.datasets import make_classification

    X, y = make_classification(n_samples=100, n_features=2,
random_state=42)

    trained_model = train_model(X, y)

    print(f"Model type: {type(trained_model)}")
```

3. Verifica el estado de los archivos modificados:

```
Unset
git status

# Observa que data_ingestion.py y model_training.py están como
Changes not staged for commit (Modified).
```

4. Usa `git diff` para ver qué cambios has realizado en el directorio de trabajo (sin staged):

```
Unset
git diff

# Observa las líneas con '+' indicando las adiciones en Python.
```

Paso 6: Staging Selectivo y Segundo `Commit`

1. Solo añade `data_ingestion.py` al área de staging:

```
Unset
git add data_ingestion.py
```

2. Verifica el estado:

```
Unset
git status

# data_ingestion.py está Staged, model_training.py está Modified.
```

3. Usa `git diff --staged` para ver solo los cambios en el área de staging:

```
Unset
git diff --staged

# Verás el contenido de data_ingestion.py que añadiste.
```

4. Ahora, usa `git diff` sin argumentos para ver solo lo que NO está en staging:

```
Unset
git diff

# Verás el contenido de model_training.py que añadiste.
```

5. Realiza un commit solo con los cambios de `data_ingestion.py`:

```
Unset
git commit -m "feat: Implementar carga de datos y simulación de CSV"
```

6. Verifica el estado:

```
Unset
git status

# model_training.py debería seguir como Modified (no commiteado).
```

7. Añade `model_training.py` y commitea:

```
Unset
git add model_training.py

git commit -m "feat: Añadir esqueleto para entrenamiento de modelo
con scikit-learn"
```

Paso 7: Visualizar el Historial Completo (`git log`)

1. Mira el historial completo de tus commits:

```
Unset
git log
```

2. Mira el historial en formato corto y gráfico:

```
Unset
git log --oneline --graph

# Observa la secuencia lineal de tus commits.
```

Paso 8: Deshaciendo Cambios (Antes de Commitear)

1. Crea un nuevo archivo de resultados temporal:

```
Unset
touch experimental_results.txt

echo "Resultados del experimento A" > experimental_results.txt
```

2. Modifica `README.md` con un error deliberado (ej. borra una línea importante):

```
Unset
# Abre README.md con tu editor y borra o cambia algo.

# Por ejemplo, deja solo "Mi Proyecto IA" y borra el resto si lo
tenía.
```

```
echo "Mi Proyecto IA" > README.md
```

3. Verifica el estado:

```
Unset
git status

# experimental_results.txt aparecerá como Untracked.

# README.md aparecerá como Modified.
```

4. Simula que te arrepientes de este cambio en `README.md` y quieres descartarlo:

```
Unset
git restore README.md
```

5. Verifica que `README.md` ha vuelto a su estado original (el del último commit):

```
Unset
git status
```

6. Simula la eliminación de `experimental_results.txt` (¡importante `dry-run`!):

```
Unset
git clean -n

# Salida: Would remove experimental_results.txt
```

7. Ahora sí, elimínalo:

```
Unset
git clean -f
```

8. Verifica con `git status` que todo está limpio.

8. Posibles Errores y Soluciones

Aquí te presento algunos errores comunes que podrías encontrar en esta clase y cómo solucionarlos, en el contexto de ciencia de datos.

1. **Error:** `fatal: pathspec 'nombre_archivo.py' did not match any files`
    - **Causa:** Intentaste `git add` o `git restore` un archivo que no existe en tu directorio de trabajo, o el nombre del archivo está mal escrito. También ocurre si intentas `git restore` un archivo `Untracked` (Git no tiene una versión anterior para restaurar).
    - **Solución:** Verifica que el nombre del archivo sea correcto. Si es un archivo `Untracked` y quieres eliminarlo, usa `git clean`. Si quieres añadirlo para que Git lo rastree, asegúrate de que existe y usa `git add`.
2. **Error:** `fatal: no changes added to commit (use "git add" and/or "git commit -a")`
    - **Causa:** Intentaste `git commit`, pero no hay cambios en el Área de Staging. O si intentaste `git commit -a`, pero no hay archivos `Modified` (que ya estuvieran rastreados) para incluir.
    - **Solución:** Primero, usa `git status` para ver qué cambios tienes.
        - Si hay scripts `Modified` o nuevos notebooks `Untracked`, usa `git add .` para moverlos al staging.
        - Si solo hay scripts `Modified` (no nuevos), puedes usar `git commit -am "Mensaje"`.
3. **Error:** `On branch main / Your branch is up to date with 'origin/main'. / nothing to commit, working tree clean` **después de hacer cambios.**
    - **Causa:** Has modificado scripts o notebooks, pero Git no los reconoce como cambios pendientes. Esto generalmente significa que no guardaste los archivos en tu editor, o que los cambios son solo de espacios en blanco/comentarios y Git (o tu configuración) los ignora por defecto.
    - **Solución:** Asegúrate de haber guardado los cambios en tu editor. Vuelve a revisar `git status`.
4. **Error:** `fatal: clean.requireForce defaults to true and neither -i, -n, nor -f given; refusing to clean`
    - **Causa:** Intentaste `git clean` sin especificar cómo quieres que se comporte (simular, forzar, interactuar). Git te protege para que no borres archivos accidentalmente, especialmente útil para no eliminar accidentalmente un dataset.
    - **Solución:** Siempre usa `git clean -n` (simular) primero para ver qué se borraría. Luego, si estás seguro, usa `git clean -f` (forzar) o `git clean -i`

(interactivo). Si también quieres borrar directorios de resultados, añade `-d` (ej. `git clean -fd`).

Conclusión de la Clase 2

¡Excelente trabajo, científicos de datos! En esta clase, hemos solidificado nuestra comprensión del ciclo de vida de los archivos en Git, aprendiendo a mover tus scripts y notebooks de un estado a otro con comandos clave como `git add` y `git commit`. También hemos explorado `git status` para mantenernos informados y `git diff` para visualizar las diferencias en tus experimentos. Además, hemos visto cómo "deshacer" cambios de forma segura antes de que se conviertan en parte del historial de tu proyecto.

Estás construyendo una base sólida. En la próxima clase, daremos un salto importante al aprender sobre las **ramas**, una de las características más poderosas de Git para el desarrollo paralelo y la experimentación en ciencia de datos.