Clase 6: Buenas Prácticas en el Uso de Git y Configuración SSH

Objetivos de la Clase

Al finalizar esta clase, serás capaz de:

- Configurar y comprender la conexión **SSH** con plataformas como GitHub o GitLab para una interacción segura y eficiente.
- Aplicar **convenciones y estándares** para mensajes de commit y nombres de ramas que mejoren la claridad y trazabilidad de tu proyecto.
- Utilizar el archivo `.gitignore` de forma efectiva para excluir archivos innecesarios o sensibles de tu repositorio de datos.
- Comprender cuándo y cómo **restaurar cambios de forma segura** utilizando `git revert`, `git reset`, y `git checkout`.

1. Configurando la Conexión SSH con Plataformas Remotas

La autenticación SSH (`Secure Shell`) es un protocolo de red criptográfico que permite una comunicación segura entre dos dispositivos conectados a una red, como tu computadora local y un servidor remoto (ej. GitHub). Configurar SSH te permite interactuar con tus repositorios remotos sin tener que introducir tu nombre de usuario y contraseña repetidamente.

1.1. ¿Por qué usar SSH en Git?

- **Comodidad:** Una vez configurado, no necesitas introducir tus credenciales para cada `push` o `pull`.
- **Seguridad:** Utiliza un par de claves criptográficas (pública y privada) en lugar de una contraseña, lo que es generalmente más seguro.
- **Automatización:** Es esencial para scripts de automatización (como CI/CD) que necesitan interactuar con repositorios remotos.

1.2. ¿Cómo funciona SSH en Git?

1. Generas un **par de claves SSH**: una clave privada (que se guarda en tu máquina y nunca se comparte) y una clave pública (que puedes compartir libremente).
2. Añades tu **clave pública** a tu perfil en la plataforma remota (GitHub, GitLab, Bitbucket).

3. Cuando intentas conectarte al remoto (ej. con `git push`), tu cliente SSH usa tu **clave privada** para autenticarse con el servidor, que verifica que coincida con la clave pública que tú le proporcionaste.

[Image of SSH Connection Flow]

1.3. Generar una Clave SSH

Paso a Paso:

1. **Abre tu terminal.**
2. **Verifica si ya tienes claves SSH existentes.** Las claves SSH se suelen almacenar en el directorio `~/.ssh/` (el `~` representa tu directorio de usuario).

```
Unset

ls -al ~/.ssh
```

3. Si ves archivos como `id_rsa`, `id_rsa.pub`, `id_ed25519`, `id_ed25519.pub`, etc., ya tienes claves. Si ves un archivo con extensión `.pub` (pública) y otro sin ella (privada), ya tienes un par. Puedes usar una existente o generar una nueva. Si decides usar una existente, salta al paso 1.5.
4. **Genera un nuevo par de claves SSH.** Se recomienda el algoritmo `ed25519` por ser más seguro y rápido, pero `rsa` también es válido.

```
Unset

# Usando ed25519 (recomendado)
ssh-keygen -t ed25519 -C "tu.email.cientifico@example.com"

# O usando RSA si ed25519 no está disponible/preferido
# ssh-keygen -t rsa -b 4096 -C "tu.email.cientifico@example.com"
```

    - `-t`: Tipo de algoritmo.
    - `-b 4096`: Tamaño de la clave (solo para RSA).
    - `-C`: Comentario para identificar la clave (usa tu email).
5. **Guarda la clave y establece una frase de contraseña.**

- Te preguntará dónde guardar la clave (`Enter a file in which to save the key (/Users/tu_usuario/.ssh/id_ed25519):`). Presiona `Enter` para aceptar la ubicación predeterminada (recomendado).
- Luego te pedirá una `passphrase` (frase de contraseña). **Es muy recomendable establecer una frase de contraseña** robusta. Esta se te pedirá la primera vez que uses la clave en una sesión (o si tu agente SSH no está ejecutándose).

6. **Inicia el agente SSH y añade tu clave.** El agente SSH mantiene tus claves en memoria para que no tengas que introducir la `passphrase` cada vez.

```
Unset

eval "$(ssh-agent -s)" # Inicia el agente SSH
ssh-add ~/.ssh/id_ed25519 # Añade tu clave (ajusta el nombre si es diferente)
```

- En macOS, puedes usar `ssh-add -K ~/.ssh/id_ed25519` para añadirla al llavero y que persista entre reinicios.

1.4. Añadir tu Clave Pública a GitHub (o GitLab/Bitbucket)

Ahora que tienes tu clave pública, necesitas dársela a la plataforma remota.

Paso a Paso:

1. **Copia tu clave pública al portapapeles.**
   - **macOS:**

```
Unset

pbcopy < ~/.ssh/id_ed25519.pub
```

   - **Windows (Git Bash):**

```
Unset

cat ~/.ssh/id_ed25519.pub | clip
```

- **Linux (requiere `xclip`):**

```
Unset

sudo apt-get install xclip # Instalar si no lo tienes
xclip -sel clip < ~/.ssh/id_ed25519.pub
```

- Si no tienes estas herramientas o la ruta es diferente, simplemente abre el archivo `~/.ssh/id_ed25519.pub` (o el nombre de tu clave) con un editor de texto y copia todo su contenido.

2. **Abre la configuración de SSH en tu plataforma remota:**
   - **GitHub:** Ve a `Settings` > `SSH and GPG keys` (https://github.com/settings/keys). Haz clic en `New SSH key` o `Add SSH key`.
   - **GitLab:** Ve a `User Settings` > `SSH Keys` (https://gitlab.com/-/profile/keys).
   - **Bitbucket:** Ve a `Personal settings` > `SSH keys`.
3. **Pega tu clave pública.**
   - Dale un **Título descriptivo** (ej. "Mi Laptop de Análisis de Datos").
   - Pega el contenido de tu clave pública en el campo "Key".
   - Haz clic en `Add SSH key`.

1.5. Probar la Conexión SSH

Para asegurarte de que todo funciona correctamente:

```
Unset
ssh -T git@github.com
# O para GitLab: ssh -T git@gitlab.com
```

- Si ves un mensaje como: `Hi tu_usuario! You've successfully authenticated, but GitHub does not provide shell access.`, ¡la configuración es exitosa!
- Si te pide una frase de contraseña, introdúcela. Si te la pide repetidamente, verifica que tu agente SSH esté funcionando.

2. Buenas Prácticas para Mensajes de Commit

Los mensajes de commit son el **diario de tu proyecto de datos**. Un buen mensaje de commit debe explicar qué cambios hiciste y, lo más importante, **por qué** los hiciste. Esto es crucial para la reproducibilidad, la depuración y la colaboración.

2.1. ¿Cada cuánto debería hacer un commit?

- **Frecuentemente y en pequeños incrementos.** Es mejor hacer muchos commits pequeños y lógicos que pocos commits grandes.
- Cada commit debe representar una **unidad lógica de trabajo** (ej., "implementar función de limpieza `fillna`", "entrenar modelo con nuevos hiperparámetros", "añadir script de visualización de distribuciones").
- **Beneficios:** Fácil reversión, mejor trazabilidad, simplifica la revisión de código.

2.2. Reglas para Escribir un Buen Mensaje de Commit (Convenciones Semánticas)

Un buen mensaje de commit generalmente sigue una estructura y ciertas reglas para hacerlo legible y útil. La convención de "Conventional Commits" es ampliamente adoptada.

`[Image of Good Commit Message Examples]`

Estructura: `<tipo>(<alcance>): <descripción_corta>`

Reglas:

1. **Línea de Asunto (50-72 caracteres max):**
    - **Usar el verbo imperativo:** Escribe como si le estuvieras dando una orden al repositorio. Ej., "Añadir", "Corregir", "Refactorizar", "Implementar".
    - **Sé conciso y directo:** Describe qué hizo el commit de manera clara y breve.
    - **Sin punto final:** La línea de asunto no lleva punto al final.
    - **Mayúscula inicial:** La primera letra de la descripción debe ser mayúscula.
2. **Cuerpo del Mensaje (Opcional, pero recomendado para contexto):**
    - Deja una línea en blanco después de la línea de asunto.
    - Proporciona más contexto: **Por qué** se hicieron los cambios, los supuestos, los resultados relevantes, las implicaciones.
    - Puede incluir referencias a issues (ej., `Fixes #123`).
    - Usa el formato de texto normal (puntuación, mayúsculas/minúsculas según la gramática).

Tipos de Commit Comunes (para `<tipo>`):

- `feat`: Una nueva característica o funcionalidad (ej., "feat: Añadir script para entrenamiento de modelo de regresión").

- `fix`: Una corrección de un bug (ej., "fix: Corregir error en el filtro de datos para valores nulos").
- `docs`: Cambios en la documentación (ej., "docs: Actualizar README con instrucciones de ejecución").
- `style`: Cambios que no afectan el significado del código (formato, espacios, punto y coma, etc.) (ej., "style: Formatear scripts con Black").
- `refactor`: Un cambio en el código que no añade funcionalidad ni corrige un bug, pero mejora la estructura o legibilidad (ej., "refactor: Reestructurar funciones de preprocesamiento").
- `test`: Añadir o corregir tests (ej., "test: Añadir pruebas unitarias para función de normalización").
- `chore`: Mantenimiento general, actualizaciones de dependencias, etc. (ej., "chore: Actualizar versión de scikit-learn a 1.2.0").
- `perf`: Mejoras de rendimiento (ej., "perf: Optimizar carga de dataset grande").
- `build`: Cambios que afectan el sistema de build o dependencias externas (ej., "build: Actualizar Dockerfile con nueva versión de Python").
- `ci`: Cambios en la configuración de CI/CD (ej., "ci: Añadir paso para ejecutar tests en GitLab CI").

Ejemplos de Mensajes de Commit para Científicos de Datos:

- `feat(model): Implementar algoritmo de clustering K-Means`
  - `Este commit introduce el algoritmo K-Means para la segmentación de clientes. Se ha añadido la lógica en 'src/clustering.py' y se ha integrado en el pipeline de entrenamiento.`
- `fix(data): Corregir manejo de valores NaN en dataset de ventas`
  - `Se ha modificado 'data_clean.py' para reemplazar los valores NaN en la columna 'precio' con la media. Antes, esto causaba errores en el entrenamiento del modelo.`
- `docs: Actualizar README con instrucciones para ambiente Conda`
- `refactor(pipeline): Modularizar funciones de preprocesamiento`
- `perf(ingestion): Optimizar lectura de archivos Parquet`

2.3. Herramientas para Validar Commits (Husky, Commitlint)

Puedes automatizar la validación de tus mensajes de commit usando **Git Hooks** (lo veremos en la Clase 8) con herramientas como:

- **Husky:** Permite ejecutar scripts predefinidos antes de ciertos eventos de Git (ej. `pre-commit`, `commit-msg`).

- **Commitlint:** Valida que tus mensajes de commit sigan una convención específica (ej. Conventional Commits).

3. Convenciones de Nombres para Ramas

Los nombres de las ramas también deben ser descriptivos y seguir una convención para mantener el repositorio ordenado.

3.1. Consejos para Nombrar Ramas

1. **Sé consistente:** Usa siempre el mismo patrón. Documenta la convención para tu equipo.
2. **Nombres concisos y descriptivos:** Deben indicar el propósito de la rama.
3. **Usa guiones (`-`) para separar palabras:** Evita espacios.
4. **Minúsculas:** Generalmente se usan letras minúsculas.
5. **Prefijos de tipo:** Similar a los commits, para indicar el tipo de trabajo.
    - `feat/`: Para una nueva funcionalidad o experimento (ej., `feat/analisis-segmentacion-clientes`).
    - `bugfix/` o `fix/`: Para correcciones de errores (ej., `bugfix/error-carga-csv`).
    - `hotfix/`: Para correcciones urgentes en producción (ej., `hotfix/modelo-cargando-mal`).
    - `refactor/`: Para refactorizaciones del código (ej., `refactor/modularizar-preprocesamiento`).
    - `docs/`: Para cambios en la documentación (ej., `docs/actualizar-api-modelo`).
    - `exp/`: Para experimentos que quizás no se integren (ej., `exp/prueba-xgboost-vs-rf`).
6. **Incluir IDs de Tickets/Issues (si aplicas):** Si usas un sistema de gestión de proyectos (Jira, Asana, Trello), incluir el ID del ticket al inicio del nombre de la rama facilita la trazabilidad.
    - Ej., `DS-123/feat-nueva-visualizacion-outliers`
    - Ej., `BUG-456/fix-nan-imputation`

4. Creación de `.gitignore` para Proyectos de Datos

No todos los archivos de tu proyecto de ciencia de datos deben ser rastreados por Git. El archivo `.gitignore` le dice a Git qué archivos o directorios debe ignorar, evitando que se añadan accidentalmente al repositorio.

4.1. ¿Qué archivos ignorar en Ciencia de Datos?

1. **Datasets grandes o sensibles:** Los archivos de datos muy grandes (CSV, Parquet, HDF5, etc.) que pueden ser reconstruidos o descargados no deben subirse al repositorio. Además, los datos sensibles (PII) nunca deben ser versionados directamente.
    - `data/*.csv`
    - `datasets/raw_data/`
2. **Resultados intermedios y outputs de modelos:** Archivos generados durante el proceso (modelos entrenados `.pkl`, `.h5`, gráficos `.png`, archivos de log, etc.) que pueden ser generados nuevamente.
    - `models/`
    - `results/`
    - `*.pkl`
    - `*.png`
    - `*.log`
3. **Archivos de entorno y credenciales:** Variables de entorno (`.env`), claves API, credenciales de bases de datos. **¡NUNCA subas estos archivos!**
    - `.env`
    - `credentials.json`
4. **Caché y directorios temporales de Python/IDE:**
    - `__pycache__/`
    - `*.pyc`
    - `.ipynb_checkpoints/` (para Jupyter Notebooks)
    - `.vscode/` (configuración específica de VS Code)
    - `.DS_Store` (archivos de sistema de macOS)
    - `venv/` o `.venv/` (entornos virtuales de Python)
5. **Dependencias de paquetes (entornos virtuales):**
    - `env/`
    - `venv/`
    - `_venv/`
    - `dist/` (distribuciones de paquetes)
    - `build/`

4.2. ¿Cómo crear un `.gitignore`?

Crea un archivo llamado `.gitignore` en la raíz de tu repositorio y lista los patrones de archivos/directorios a ignorar.

Ejemplo de `.gitignore` para un proyecto de ciencia de datos:

Unset

```
# Entornos virtuales
.venv/
venv/
env/

# Archivos de Python
__pycache__/
*.pyc
*.log
*.sqlite3 # Opcional, si usas DBs locales

# Notebooks Jupyter
.ipynb_checkpoints/
*.ipynb_checkpoints # No necesario si lo anterior ya está

# Datos
/data/*.csv # Ignora CSVs en la carpeta data, pero no la carpeta
en sí
/data/raw/ # Ignora todo el contenido de la carpeta raw dentro de
data
!/data/README.md # Excepción: si quieres que el README.md dentro
de 'data' sea rastreado

# Resultados y modelos
/models/
/results/
*.pkl
*.h5
*.joblib
*.png
*.svg # Si son outputs de gráficos

# Archivos de sistema y editor
.DS_Store
.vscode/ # Configuración de VS Code
```

```
# Archivos de entorno y credenciales (¡MUY IMPORTANTE!)
.env
credentials.json
```

- Las líneas que empiezan con `#` son comentarios.
- Puedes usar `*` como comodín.
- Puedes usar `!` para hacer una excepción (ej., `!/data/README.md` incluye el README en la carpeta `data` a pesar de que la carpeta esté ignorada).

4.3. Dejar de Rastreado un Archivo ya Commiteado (`git rm --cached`)

Si por error commiteaste un archivo que debería haber sido ignorado (ej. un dataset grande), puedes hacer que Git deje de rastrearlo sin eliminarlo de tu directorio de trabajo.

1. **Añade el archivo al `.gitignore`** (para que no vuelva a ser rastreado).
2. **Ejecuta `git rm --cached <ruta_del_archivo>`:**

```
Unset

git rm --cached large_dataset.csv
```

3. **Haz un commit** para registrar la eliminación del rastreo:

```
Unset

git commit -m "chore: Stop tracking large_dataset.csv"
```

Ahora, `large_dataset.csv` seguirá en tu carpeta local, pero Git ya no lo gestionará.

5. Restaurar Cambios de Forma Segura

En la Clase 3, vimos `git reset` para deshacer commits locales. Ahora, hablaremos sobre cómo "deshacer" cambios de forma más segura, especialmente cuando ya has compartido tu trabajo.

5.1. `git revert`: Deshacer Commits Publicados (Seguro)

`git revert <hash_commit>` es la forma segura de deshacer un commit que ya ha sido empujado a un repositorio remoto y potencialmente compartido con otros.

- **¿Qué hace?** Crea un **nuevo commit** que aplica los cambios inversos de un commit anterior. No reescribe el historial. Mantiene el historial intacto y añade un nuevo "deshacer" en la línea de tiempo.
- **Cuándo usarlo: Siempre que quieras deshacer un commit que ya ha sido publicado** y compartido.
- **Ventaja:** Mantiene la integridad del historial del repositorio compartido, evitando problemas de sincronización para otros colaboradores.

Ejemplo para Científicos de Datos: Imagina que un commit anterior (`abcdef1`) introdujo una función en `model_evaluation.py` que causa una regresión en la precisión del modelo.

```
Unset
# Paso 1: Identifica el hash del commit que quieres deshacer
git log --oneline
# ...
# abcdef1 feat: Añadir función de evaluación de precisión (este
tiene el bug)
# ...

# Paso 2: Deshace el commit con revert
git revert abcdef1
# Git abrirá tu editor para el mensaje del nuevo commit (ej.
"Revert 'feat: Añadir función de evaluación de precisión'")
# Guarda y cierra el editor.
```

Ahora verás un nuevo commit en tu historial que "deshace" los cambios de `abcdef1`. Puedes empujar este nuevo commit sin problemas.

5.2. `git reset`: Revertir el Historial (¡Cuidado en Repositorios Compartidos!)

Como vimos, `git reset` reescribe el historial al mover el puntero de la rama.

- **`git reset --soft <hash>`:** Mueve el HEAD de la rama, pero los cambios deshechos permanecen en staging.
- **`git reset --mixed <hash>`:** Mueve el HEAD, los cambios deshechos se mueven al directorio de trabajo.
- **`git reset --hard <hash>`:** Mueve el HEAD y descarta los cambios deshechos (¡Pérdida de datos!).
- **Cuándo usarlo: Solo para deshacer commits que aún no han sido publicados** en un remoto o compartidos con otros. Usarlo en un historial compartido causaría divergencias y problemas para tus colaboradores.

5.3. `git checkout <commit_hash> -- <ruta_archivo>`: Recuperar un Archivo Específico

Si solo quieres recuperar una versión anterior de un archivo específico sin afectar el resto de tu historial o tu rama, `git checkout` es útil.

- **¿Qué hace?** Copia la versión del archivo desde un commit específico (o rama) a tu Directorio de Trabajo. El archivo pasará al estado `Modified` (o `Staged` si ya estaba rastreado).
- **Cuándo usarlo:** Cuando necesitas una versión antigua de un script o un archivo de configuración para referencia o para reintroducir un cambio específico.

Ejemplo para Científicos de Datos: Necesitas la versión de `data_loader.py` de un commit anterior (`cba9876`) para comparar su rendimiento.

```
Unset
git checkout cba9876 -- data_loader.py
# El archivo data_loader.py en tu directorio de trabajo se
actualizará a la versión de ese commit.
# Estará en estado "Modified" o "Staged" (dependiendo de su
estado previo).
# Puedes luego hacer un commit si quieres guardar esta versión.
```

6. Actividad Práctica (Laboratorio 6)

Vamos a aplicar las buenas prácticas aprendidas.

Paso 1: Preparación del Entorno y Configuración SSH

1. Si no lo hiciste en la Clase 5, crea un nuevo repositorio en GitHub (vacío, sin README ni .gitignore).
2. Si no tienes claves SSH configuradas, síguelo en la sección 1 de esta clase. **¡Asegúrate de que `ssh -T git@github.com` funciona antes de continuar!**
3. Crea un nuevo directorio para este laboratorio y entra en él:

```
Unset

mkdir clase6-lab-ds
cd clase6-lab-ds
git init
```

4. Crea un archivo `requirements.txt` y `main_script.py`:

```
Unset

# requirements.txt
pandas==1.5.0
scikit-learn==1.2.0
matplotlib==3.6.0
```python
# main_script.py
import pandas as pd

def process_data(df):
    return df.fillna(0)

if __name__ == "__main__":
    df = pd.DataFrame({'col1': [1, None]})
    processed_df = process_data(df)
    print(processed_df)
```

5. Haz el primer commit usando un mensaje semántico:
```

```
Unset

git add .
git commit -m "feat: Initial setup with data processing script"
```

6. Añade el remoto y haz el primer `push` con SSH:

```
Unset

git remote add origin
git@github.com:tu-usuario/nombre-repo-clase6.git
git push -u origin main
```

7. Si te pide passphrase, introdúcela. Si falla, revisa tu configuración SSH.

Paso 2: Práctica de Mensajes de Commit y Nombres de Rama

1. **Crea una rama para una nueva funcionalidad:**

```
Unset

git switch -c feat/nueva-visualizacion-eda
```

2. Crea un archivo `eda_visuals.py` y añade contenido:

```
Unset

# eda_visuals.py
import matplotlib.pyplot as plt
import seaborn as sns

def plot_distribution(data, column):
    plt.figure(figsize=(8, 6))
    sns.histplot(data[column], kde=True)
    plt.title(f'Distribution of {column}')
```

```
    plt.show()

if __name__ == "__main__":
    # Ejemplo de uso
    import pandas as pd
    df = pd.DataFrame({'value': [10, 12, 15, 12, 18, 20]})
    plot_distribution(df, 'value')
```

3. Realiza el commit con un **mensaje semántico y descriptivo**:

```
Unset

git add eda_visuals.py
git commit -m "feat(eda): Implement función para graficar
distribución de columnas"
```

4. Modifica `main_script.py` (ej. cambia el print) y `requirements.txt` (ej. actualiza `pandas==2.0.0`)
5. Realiza un commit para cada cambio, demostrando **commits pequeños y atómicos**:

```
Unset

git add main_script.py
git commit -m "refactor: Ajustar mensaje de salida en
main_script"

git add requirements.txt
git commit -m "chore: Actualizar pandas a la versión 2.0.0"
```

6. Envía esta rama con todos sus commits al remoto:

```
Unset

git push origin feat/nueva-visualizacion-eda
```

7. *En GitHub, ve y verifica los commits en la rama. Observa la claridad de los mensajes.*

Paso 3: Uso de `.gitignore`

1. Crea una carpeta `temp_outputs/` y dentro un archivo `temp_model.pkl`:

```
Unset

mkdir temp_outputs
touch temp_outputs/temp_model.pkl
```

2. Crea un archivo `secrets.env` (que contendrá credenciales sensibles simuladas):

```
Unset

echo "API_KEY=your_secret_key" > secrets.env
```

3. Verifica `git status`. `temp_outputs/` y `secrets.env` deberían estar como `Untracked`.
4. Crea o edita tu archivo `.gitignore` en la raíz de tu repositorio y añade:

```
Unset

# Archivos y directorios a ignorar
temp_outputs/
secrets.env
```

5. Guarda `.gitignore`. Verifica `git status` de nuevo. `temp_outputs/` y `secrets.env` deberían haber desaparecido de la lista de `Untracked`.
6. Añade `.gitignore` y haz commit:

```
Unset

git add .gitignore
git commit -m "chore: Add .gitignore for temporary outputs and
secrets"
```

7. Envía este commit a tu rama:

```
Unset

git push origin feat/nueva-visualizacion-eda
```

Paso 4: Deshacer Cambios con `git revert` (Simulando un Commit Publicado)

1. **En GitHub.com**: Abre una Pull Request desde `feat/nueva-visualizacion-eda` a `main`. Fusiónala (con `Merge pull request`). Esto simulará que tus cambios se publicaron. Luego, elimina la rama remota.
2. **En tu Terminal Local**:
    - Vuelve a la rama `main` y haz `pull` para tener los cambios de la PR fusionada:

```
Unset

git switch main
git pull origin main
```

    - Identifica el hash del commit que introdujo la función de `plot_distribution` (o cualquier commit de tu rama que ya se fusionó). Puedes usar `git log --oneline`.

```
Unset

git log --oneline
# Busca el commit del tipo "feat(eda): Implement función para
graficar distribución..."
# Copia su hash (ej., a1b2c3d)
```

- **Simula que ese commit (a1b2c3d) causó un problema y quieres deshacerlo de forma segura.**

```
Unset

git revert <hash_del_commit_a_revertir>
# Se abrirá el editor para que confirmes el mensaje del nuevo commit de reversión. Guarda y cierra.
```

- Verifica `git log --oneline --graph`. Verás que se ha añadido un nuevo commit que deshace los cambios del commit original.
- Envía este nuevo commit al remoto:

```
Unset

git push origin main
```

7. Posibles Errores y Soluciones

1. **Error: `Permission denied (publickey)` o `Authentication failed` al usar SSH.**
   - **Causa:** Tu clave pública SSH no está correctamente registrada en tu perfil de GitHub/GitLab, o tu clave privada no está en tu máquina, no está añadida al agente SSH, o la `passphrase` es incorrecta.
   - **Solución:**
     - Verifica que la clave SSH se haya generado correctamente (`ls -al ~/.ssh/`).
     - Asegúrate de haber añadido la clave pública a tu perfil en la plataforma remota.
     - Verifica que el agente SSH esté corriendo (`eval "$(ssh-agent -s)"`) y que tu clave esté añadida (`ssh-add ~/.ssh/id_ed25519`).
     - Si es la primera vez que usas la clave en la sesión, te pedirá la `passphrase`. Asegúrate de introducirla correctamente.
     - Probar la conexión con `ssh -T git@github.com`.
2. **Error: `fatal: not a git repository (or any of the parent directories): .git` al intentar `git remote add`.**
   - **Causa:** Estás intentando añadir un remoto a un directorio que no ha sido inicializado como repositorio Git.

    - **Solución:** Asegúrate de ejecutar `git init` en tu directorio de proyecto antes de añadir el remoto.
3. **Error: `Your local changes to the following files would be overwritten by checkout:` (o `git switch`).**
    - **Causa:** Tienes cambios sin commitear en tu directorio de trabajo o en staging, y al cambiar de rama, Git detecta que esos cambios serían sobrescritos.
    - **Solución:**
        - **Opción A (Recomendada):** Haz un `commit` de tus cambios antes de cambiar de rama.
        - **Opción B:** Si los cambios son temporales y no quieres commitearlos, usa `git stash` (lo veremos en la Clase 8) para guardarlos temporalmente.
        - **Opción C (¡Solo si estás seguro de descartar!):** Deshaz tus cambios con `git restore .` antes de cambiar de rama.
4. **`git status` sigue mostrando archivos ignorados después de crear `.gitignore`.**
    - **Causa:**
        - El archivo `.gitignore` no está en la raíz correcta del repositorio.
        - El patrón en `.gitignore` es incorrecto.
        - El archivo ya fue rastreado (commiteado) antes de que se añadiera al `.gitignore`.
    - **Solución:**
        - Verifica la ubicación y los patrones en `.gitignore`.
        - Si el archivo ya fue rastreado, usa `git rm --cached <ruta_archivo>` y luego haz un commit para dejar de rastrearlo.
5. **Confusión entre `git revert` y `git reset`.**
    - **Causa:** Ambos "deshacen" cambios, pero de formas fundamentalmente diferentes.
    - **Solución:**
        - Recuerda: **`git revert` crea un *nuevo commit* que deshace los cambios (seguro para historial publicado).**
        - **`git reset` *reescribe el historial* (peligroso para historial publicado, solo usar localmente).**
        - Prioriza `git revert` para deshacer cambios que ya están en el repositorio remoto.

Conclusión de la Clase 6

¡Excelente trabajo, científicos de datos! Hoy hemos profundizado en la seguridad y la profesionalidad de tu flujo de trabajo con Git. Has aprendido a configurar la conexión **SSH**, una herramienta fundamental para una interacción eficiente y segura con tus repositorios remotos. Además, hemos establecido bases sólidas en **buenas prácticas**: cómo redactar mensajes de commit claros, nombrar ramas de forma consistente y utilizar `.gitignore` para mantener tus

proyectos limpios. Finalmente, dominas las técnicas para **deshacer cambios de forma segura**, lo cual es crucial en entornos colaborativos.

Con estas habilidades, estás mucho mejor preparado para trabajar en equipo y mantener un historial de proyecto impecable. En la próxima clase, exploraremos las diferentes **estrategias de flujo de trabajo** que los equipos de desarrollo utilizan con Git para organizar su colaboración.