Clase 8: Git Avanzado y Herramientas Útiles (Parte 1)

Objetivos de la Clase

Al finalizar esta clase, serás capaz de:

- Utilizar **`git stash`** para almacenar temporalmente cambios no commiteados y recuperarlos posteriormente.
- Crear y gestionar **etiquetas (`tags`)** para marcar puntos importantes en el historial de tu proyecto (ej., versiones de modelos).
- Aplicar selectivamente commits de una rama a otra utilizando **`git cherry-pick`**.
- Introducir el concepto de **Git Hooks** para automatizar tareas en tu flujo de trabajo.
- Realizar **análisis de historial** con `git blame` y `git bisect` para depuración de código y datos.
- Comprender brevemente el uso de **submódulos** y la integración con herramientas externas.

1. `git stash`: El Almacén Temporal de Cambios

Imagina que estás trabajando en un script de preprocesamiento de datos muy complejo en tu rama actual, y de repente surge una emergencia (un bug crítico en producción o una petición urgente de un análisis rápido). No quieres commitear tu trabajo a medias, pero tampoco quieres perderlo. Aquí es donde `git stash` es tu mejor aliado.

1.1. ¿Qué es `git stash`?

`git stash` guarda temporalmente todos los cambios de tu Directorio de Trabajo y de tu Área de Staging que no están commiteados, dejándote con un directorio de trabajo limpio. Es como poner tu trabajo actual "en pausa" o "en un cajón" para poder cambiar de contexto o rama.

1.2. Guardar Cambios en un Stash

- **`git stash`** o **`git stash push`**: Guarda tus cambios modificados y stasheados.
    - Por defecto, **no guarda archivos `Untracked`** (nuevos archivos que no han sido `git add`eados).
    - **`git stash -u`** o **`--include-untracked`**: Incluye archivos `Untracked`.

- **`git stash -a`** o **`--all`**: Incluye todos los archivos (modificados, stasheados, `Untracked` e ignorados por `.gitignore`). Útil para limpiezas completas.
- **`git stash push -m "Mensaje descriptivo"`**: Añade un mensaje al stash para recordarte qué guardaste (muy recomendado).

Ejemplo para Científicos de Datos:

```
Unset
# Estás en 'feat/analisis-exploratorio' y tienes cambios a medias
# Modificas 'eda_script.py' y creas 'temp_visual.ipynb'
(untracked)

git status
# Salida:
# M eda_script.py
# ?? temp_visual.ipynb

# Guardas tus cambios temporalmente
git stash push -m "WIP: Análisis exploratorio inicial de
correlaciones"
# Salida: Saved working directory and index state On
feat/analisis-exploratorio: ...
```

Ahora tu directorio de trabajo está limpio, puedes cambiar de rama, y `temp_visual.ipynb` también se habrá guardado.

1.3. Gestionar Stashes

- **`git stash list`**: Muestra la lista de stashes guardados. Los stashes se organizan como una pila (el más reciente es `stash@{0}`).

```
Unset

git stash list
# Salida:
```

```
# stash@{0}: On feat/analisis-exploratorio: WIP: Análisis
exploratorio inicial de correlaciones
# stash@{1}: On main: Ajuste rápido en config.py
```

- **`git stash show [stash@{n}]`**: Muestra el diff del stash (qué cambios contiene). Por defecto, el último stash.
- **`git stash drop [stash@{n}]`**: Elimina un stash específico de la pila.
- **`git stash clear`**: Elimina **todos** los stashes. **¡Cuidado, no hay confirmación!**

1.4. Aplicar Cambios del Stash

- **`git stash pop [stash@{n}]`**: Aplica los cambios del stash a tu Directorio de Trabajo y Área de Staging, y luego **elimina el stash de la pila**. Por defecto, aplica el `stash@{0}`.
- **`git stash apply [stash@{n}]`**: Aplica los cambios del stash, pero **mantiene el stash en la pila**. Útil si quieres aplicar el mismo conjunto de cambios en varias ramas o contextos.

Ejemplo:

Unset
```
# Después de resolver la emergencia y volver a tu rama original
git switch feat/analisis-exploratorio

# Recuperas tus cambios
git stash pop
# Tus cambios vuelven al directorio de trabajo.
```

Si aplicas un stash y hay conflictos con los cambios actuales en tu directorio de trabajo, Git te lo indicará y deberás resolverlos manualmente (como en un `merge`).

1.5. Crear una Rama desde un Stash

Si te das cuenta de que los cambios en un stash son en realidad una nueva funcionalidad o bugfix que debería tener su propia rama, puedes hacer esto:

- **`git stash branch <nueva_rama_nombre> [stash@{n}]`**: Crea una nueva rama a partir del commit donde se creó el stash, aplica los cambios del stash a esa nueva rama, y luego **elimina el stash**.

```
Unset

git stash branch nueva-feature-desde-stash
```

2. `git tag`: Marcando Hitos Importantes

Las etiquetas (`tags`) en Git se utilizan para marcar puntos específicos en el historial de commits como importantes. Son como "marcadores permanentes" que no se mueven. Son ideales para versiones de software, versiones de modelos o liberaciones de datasets.

2.1. Tipos de Tags

- **`lightweight tags` (ligeras):** Son solo un puntero a un commit (como una rama que nunca se mueve). No contienen información adicional.
- **`annotated tags` (anotadas):** Son objetos completos en la base de datos de Git. Contienen el nombre del *tagger*, email, fecha, y un mensaje de tag. Son la opción recomendada para tags importantes (ej., lanzamientos).

2.2. Crear Tags

- **Crear un `annotated tag` (recomendado para hitos):**

```
Unset

git tag -a v1.0.0 -m "Versión 1.0.0 del modelo de recomendación
de productos"
```

    - `-a`: Crea un tag anotado.
    - `-m`: Añade un mensaje al tag.
- **Crear un `lightweight tag`:**

```
Unset

git tag v1.0-data-release
```

- **Etiquetar un commit específico (no el HEAD):** Puedes especificar el hash del commit al que quieres aplicar la etiqueta.

```
Unset

git tag -a v1.0.0-bugfix -m "Corrección de bug en v1.0.0"
abcdef123 # abcdef123 es el hash
```

2.3. Gestionar Tags

- **`git tag`**: Lista todos los tags en tu repositorio.
- **`git show <tag_nombre>`**: Muestra la información detallada de un tag (especialmente útil para `annotated tags`).
- **`git tag -d <tag_nombre>`**: Elimina un tag local.

2.4. Compartir Tags (Pushing Tags)

Los tags, por defecto, no se envían a los repositorios remotos cuando haces `git push`. Debes enviarlos explícitamente.

- **`git push origin <tag_nombre>`**: Envía un tag específico al remoto.
- **`git push origin --tags`**: Envía **todos** tus tags locales al remoto.

Ejemplo para Científicos de Datos:

```
Unset
# Después de un commit que representa una versión estable de tu
modelo
git tag -a v1.0.0-modelo-final -m "Modelo de predicción de churn,
versión final para despliegue"

# Envía el tag al repositorio remoto
git push origin v1.0.0-modelo-final
```

Ahora, cualquier miembro del equipo puede ver y clonar esa versión específica del modelo.

3. `git cherry-pick`: Aplicar Commits Selectivamente

`git cherry-pick` es una herramienta poderosa que te permite tomar commits individuales de cualquier parte del historial de una rama y aplicarlos como nuevos commits en tu rama actual. Es como "recoger" una "cereza" (un commit) y colocarla en tu propio árbol.

3.1. ¿Qué es `git cherry-pick`?

- **Propósito:** Aplicar los cambios de uno o varios commits específicos de una rama a la rama actual, sin fusionar todo el historial de la rama de origen. Crea **nuevos commits** con los mismos cambios.
- **Sintaxis:**

```
Unset

git cherry-pick <hash_commit_1> [hash_commit_2 ...]
```

- Puedes especificar uno o varios hashes de commits.

3.2. Casos de Uso para Científicos de Datos:

- **Bugfix Rápido:** Un colega en una rama de experimentación encontró y corrigió un bug en un script común. Puedes `cherry-pick` ese commit de corrección directamente a tu rama `main` sin tener que esperar a que se fusione toda la rama de experimentación.
- **Funcionalidad Pequeña:** Un cambio pequeño y autocontenido (ej. una nueva métrica, una función de visualización) está en una rama, y lo necesitas en tu rama antes de que toda la rama se fusione.
- **Backporting:** Aplicar una corrección de un bug de una versión más reciente a una versión anterior.

3.3. Consideraciones

- **Crea nuevos commits:** `cherry-pick` no mueve el commit original, sino que crea una copia con los mismos cambios, pero un nuevo hash.

- **Conflictos:** Si el commit seleccionado entra en conflicto con tu rama actual, deberás resolverlos manualmente (como en un `merge`).
- **Historial no lineal:** Usar `cherry-pick` en exceso puede hacer que el historial se vuelva confuso, ya que los mismos cambios pueden aparecer en diferentes lugares con distintos hashes. Prefiere `merge` o `rebase` cuando sea posible para integrar ramas completas.

Ejemplo:

```
Unset
# Estás en 'main'. Un commit 'cba9876' en
'bugfix/correccion-data' corrigió un error crucial.
git switch main
git cherry-pick cba9876
# Git creará un nuevo commit en 'main' con los cambios de
'cba9876'.
```

4. Introducción a Git Hooks: Automatización de Tareas

Los **Git Hooks** (o "ganchos") son scripts personalizados que Git ejecuta automáticamente en respuesta a ciertos eventos importantes en el ciclo de vida del repositorio. Son una forma poderosa de automatizar tareas y aplicar políticas.

4.1. ¿Qué son los Git Hooks?

Son simples archivos ejecutables (scripts de shell, Python, Ruby, etc.) ubicados en la carpeta oculta `.git/hooks/` de tu repositorio. Cuando ocurre un evento particular (ej., antes de un commit, después de un push), Git busca un script con el nombre correspondiente en esa carpeta y lo ejecuta.

4.2. Tipos de Hooks (Lado del Cliente)

Nos enfocaremos en los hooks del lado del cliente, que se ejecutan en tu máquina local.

- **`pre-commit`**: Se ejecuta **antes** de que Git pida el mensaje de commit. Si este hook sale con un error, el commit es abortado.
    - **Usos en Ciencia de Datos:** Ejecutar un linter de Python (ej., `flake8`, `pylint`), verificar que los notebooks Jupyter están limpios (sin outputs grandes), ejecutar tests unitarios rápidos.

- **`commit-msg`**: Se ejecuta **después** de que el usuario ha escrito el mensaje del commit, pero antes de que el commit se grabe. Si sale con error, el commit es abortado.
    - **Usos en Ciencia de Datos:** Validar el formato del mensaje de commit (ej., si sigue la convención de Conventional Commits), verificar la longitud del mensaje.
- **`pre-push`**: Se ejecuta **antes** de que los objetos se transfieran al repositorio remoto durante un `git push`. Si sale con error, el push es abortado.
    - **Usos en Ciencia de Datos:** Ejecutar tests de integración más largos, verificar la calidad de los datos, asegurarse de que no se suban credenciales sensibles.

4.3. Crear un Hook Simple (Ejemplo `pre-commit` con Python)

1. Navega a la carpeta `.git/hooks/` dentro de tu repositorio.

```
Unset

cd .git/hooks/
```

2. Crea un nuevo archivo con el nombre del hook que quieres implementar (ej., `pre-commit`). **Asegúrate de que no tenga extensión y sea ejecutable.**

```
Unset

touch pre-commit
chmod +x pre-commit # Hacerlo ejecutable
```

3. Edita el archivo `pre-commit` y añade tu script.
   Ejemplo (simplemente verifica un archivo de configuración):

```
Unset

#!/bin/bash

# Este script se ejecutará antes de cada commit.

echo "Running pre-commit hook..."
```

```
# Verifica si existe un archivo de configuración crítico y su
contenido.
if [ -f ../config.py ]; then
    if grep -q "production_mode = True" ../config.py; then
        echo "ERROR: No puedes commitear con production_mode =
True en config.py. Cambia a False."
        exit 1 # Aborta el commit
    else
        echo "config.py verificado. OK."
    fi
fi

exit 0 # Permite el commit
```

- `#!/bin/bash`: Shebang que indica que el script se ejecutará con Bash.
- `exit 1`: Si el script sale con un código distinto de 0, el evento de Git (commit en este caso) se aborta.
- `exit 0`: Permite que el evento continúe.

Consideraciones: Los Git Hooks son locales al repositorio y no se versionan con `git push`. Para compartirlos en un equipo, se suelen usar herramientas como **Husky** (en proyectos Node.js) o scripts de configuración en el proyecto que se encargan de copiarlos o configurarlos.

5. Análisis de Historial: Depuración de Código y Datos

Git no solo es útil para guardar versiones, sino también para depurar problemas, especialmente en proyectos de datos donde un cambio en un script puede afectar la calidad de los datos o el rendimiento del modelo.

5.1. `git blame`: ¿Quién Tocó esta Línea?

`git blame` te ayuda a determinar quién y cuándo modificó cada línea de un archivo. Es útil para rastrear la autoría de un problema o para entender por qué una línea específica tiene cierto código.

Sintaxis:

```
Unset
git blame <ruta_del_archivo>
```

Ejemplo para Científicos de Datos: Si encuentras un valor nulo inesperado en un DataFrame después de una función de limpieza, puedes usar `git blame` para ver qué commit (y por lo tanto, qué desarrollador) fue el último en modificar esa sección del script.

```
Unset
git blame data_preprocessing.py
# Salida (ejemplo simplificado):
# ^a1b2c3d (Juan Perez   2023-01-15 10:30:15 -0300  1) import
pandas as pd
# ^a1b2c3d (Juan Perez   2023-01-15 10:30:15 -0300  2)
# 4e5f6g7h (Ana Gomez    2023-03-20 14:00:00 -0300  3) def
clean_data(df):
# 8i9j0k1l (Carlos Diaz  2023-04-01 09:00:00 -0300  4)     df =
df.dropna(subset=['col_importante']) # Esta linea parece
relevante
# ...
```

Puedes filtrar por líneas específicas: `git blame -L 10,20 data_preprocessing.py`.

5.2. `git bisect`: Encontrando el Commit Culpable

`git bisect` es una herramienta de depuración que te ayuda a encontrar el commit que introdujo un bug. Utiliza una búsqueda binaria para recorrer el historial, preguntándote en cada paso si una versión específica del código tiene el bug (`bad`) o no (`good`).

Flujo de Trabajo:

1. **Iniciar `bisect`:**

```
Unset

git bisect start
```

2. **Marcar el commit `bad` (el actual, o uno que sabes que tiene el bug):**

```
Unset

git bisect bad
```

3. **Marcar un commit `good` (uno antiguo que sabes que *no* tenía el bug):**

```
Unset

git bisect good <hash_del_commit_bueno>
# O, si no sabes el hash exacto, puedes ir atrás en el tiempo:
# git bisect good HEAD~10 # 10 commits atrás
```

4. Git te moverá a un commit intermedio. En ese commit, **ejecuta tus pruebas** (ej., corre tu script de modelo, verifica la salida, etc.) y determina si el bug está presente.
    - Si el bug está presente: `git bisect bad`
    - Si el bug **no** está presente: `git bisect good`
5. Repite el paso 4 hasta que Git te diga cuál es el primer commit "malo".
6. **Salir de `bisect`:**

```
Unset

git bisect reset
```

7. Esto te devolverá a la rama y commit donde estabas antes de iniciar `bisect`.

Ejemplo para Científicos de Datos: Tu modelo de clasificación de imágenes dejó de funcionar correctamente en producción hace unos días. No sabes cuál de los últimos 20 commits lo rompió.

```
Unset
git bisect start
git bisect bad # Tu commit actual tiene el bug
git bisect good abcdef123 # Este es el commit de hace una semana,
sabes que estaba bien

# Git te moverá a un commit intermedio (ej. ghijklm)
# Ahora, en tu entorno de desarrollo, ejecuta tu pipeline de
modelo, pasa datos de prueba, etc.
# ¿El modelo funciona correctamente?
# Si funciona: git bisect good
# Si NO funciona (bug presente): git bisect bad

# Repite hasta que Git te indique el commit culpable
# Salida final: <hash_del_commit_culpable> is the first bad
commit
```

6. Trabajo con Submódulos (Breve Introducción)

Los **submódulos** permiten incluir otro repositorio Git dentro de tu repositorio principal como un subdirectorio. El submódulo mantiene su propio historial de commits y es gestionado de forma independiente.

6.1. ¿Para qué se usan los submódulos?

- **Bibliotecas Compartidas:** Si tienes un conjunto de scripts de utilidad de Python, modelos base o datasets de referencia que son proyectos de Git independientes y se utilizan en varios de tus proyectos de datos.
- **Separación de Responsabilidades:** Mantiene proyectos relacionados pero distintos separados, facilitando su gestión y desarrollo.

Ejemplo para Científicos de Datos: Imagina que tienes un repositorio principal para un proyecto de clasificación de texto (`text_classifier_project`). Dentro de él, podrías tener un submódulo para una biblioteca de utilidades de preprocesamiento de texto (`text_prep_utils`) que también usas en otros proyectos.

```
Unset
# En el directorio raíz de tu proyecto principal
git submodule add
https://github.com/tu-usuario/text_prep_utils.git
src/text_prep_utils
```

Esto creará una entrada en `.gitmodules` y clonará el repositorio `text_prep_utils` en `src/text_prep_utils`.

Consideraciones: Los submódulos pueden añadir complejidad. Actualizarlos y trabajar con ellos requiere comandos adicionales (`git submodule update`). A menudo, para bibliotecas, se prefiere la gestión de paquetes (ej., `pip` en Python).

7. Actividad Práctica (Laboratorio 8)

Vamos a practicar estos comandos avanzados en un escenario simulado.

Paso 1: Preparación del Entorno

1. Crea un nuevo directorio para este laboratorio y entra en él:

```
Unset

mkdir clase8-lab-ds
cd clase8-lab-ds
git init
```

2. Crea un archivo `analysis_script.py` y un `config.py` con contenido inicial:

```
Unset

# analysis_script.py
def analyze_data(data):
    return data.mean()
```

```
if __name__ == "__main__":
    import pandas as pd
    df = pd.DataFrame({'value': [10, 20, 30]})
    print(f"Mean: {analyze_data(df['value'])}")
```python
# config.py
PARAM_A = 10
PARAM_B = 20
```

3. Haz el primer commit:

```
Unset

git add .
git commit -m "Initial commit: Add analysis script and config"
```

4. Realiza un segundo commit en `analysis_script.py` (simulando un cambio):

```
Unset

# analysis_script.py (modificado)
def analyze_data(data):
    # Añadir un paso de procesamiento
    data = data * 2
    return data.mean()

if __name__ == "__main__":
    import pandas as pd
    df = pd.DataFrame({'value': [10, 20, 30]})
    print(f"Mean after processing: {analyze_data(df['value'])}")
```bash
git add analysis_script.py
git commit -m "feat: Add data processing step in analysis_script"
```

Paso 2: Práctica con `git stash`

1. Modifica `analysis_script.py` (ej., añade un comentario o un `print` temporal) y crea un nuevo archivo `temp_notebook.ipynb`:

```
Unset

# analysis_script.py (añade un comentario temporal)
def analyze_data(data):
    # AQUI ES NECESARIO REVISAR ALGORITMO DE NORMALIZACION
    data = data * 2
    return data.mean()
```bash
touch temp_notebook.ipynb
```

2. Verifica `git status`. Deberías ver `analysis_script.py` como `Modified` y `temp_notebook.ipynb` como `Untracked`.
3. Guarda estos cambios temporalmente usando `git stash` incluyendo los no rastreados y con un mensaje:

```
Unset

git stash push -u -m "WIP: Prueba de normalización y notebook
temporal"
```

4. Verifica `git status` (debería estar limpio) y `git stash list`.
5. Ahora, simula que cambias de rama para hacer otra cosa (no necesitas cambiar de rama físicamente, solo imagina que lo hiciste).
6. Aplica los cambios del stash de nuevo:

```
Unset

git stash pop
```

7. Verifica `analysis_script.py` y `temp_notebook.ipynb` de nuevo.

Paso 3: Práctica con `git tag` y `git cherry-pick`
```

1. Asegúrate de estar en la rama `main` y que el directorio de trabajo esté limpio.
2. Crea un `annotated tag` para el estado actual de tu proyecto, como si fuera la versión 1.0 de un análisis:

```
Unset

git tag -a v1.0.0-initial-analysis -m "Versión 1.0.0 del análisis inicial de datos"
```

3. Verifica el tag:

```
Unset

git tag
git show v1.0.0-initial-analysis
```

4. **Crea una nueva rama `bugfix/param-correction`:**

```
Unset

git switch -c bugfix/param-correction
```

5. En esta rama, modifica `config.py` para corregir un parámetro (simulando un bugfix):

```
Unset

# config.py
PARAM_A = 10
PARAM_B = 25 # Corrección: el valor correcto debería ser 25
```

6. Haz un commit de esta corrección:

```
Unset

git add config.py
git commit -m "fix: Corregir valor de PARAM_B en config.py"
```

7. Obtén el hash de este commit de corrección:

```
Unset

git log --oneline
# Copia el hash corto (ej., a1b2c3d)
```

8. Vuelve a la rama `main`:

```
Unset

git switch main
```

9. **Aplica selectivamente el commit de corrección de `bugfix/param-correction` a `main` usando `cherry-pick`:**

```
Unset

git cherry-pick <hash_del_commit_de_correccion>
# Si hay conflicto, resuélvelo y luego `git add .` y `git
commit`.
```

10. Verifica `config.py` en `main` para asegurarte de que el cambio se aplicó.
11. Elimina la rama `bugfix/param-correction` (ya no es necesaria porque el commit se "sacó" con cherry-pick):

```
Unset

git branch -d bugfix/param-correction
```

Paso 4: Creación de un `pre-commit` Hook (Ejemplo simple)

1. Navega al directorio `.git/hooks/`:

```
Unset

cd .git/hooks/
```

2. Crea el archivo `pre-commit` y hazlo ejecutable:

```
Unset

touch pre-commit
chmod +x pre-commit
```

3. Edita `pre-commit` con el siguiente contenido. Este hook simplemente verifica que no se intente commitear el archivo `config.py` si contiene una línea específica (simulando una comprobación de seguridad).

```
Unset

#!/bin/bash

# Este hook verifica que 'config.py' no tenga el modo de
producción activado.

CONFIG_FILE="../config.py"
PRODUCTION_FLAG="production_mode = True"

if [ -f "$CONFIG_FILE" ]; then
```

```
    if grep -q "$PRODUCTION_FLAG" "$CONFIG_FILE"; then
        echo
"------------------------------------------------------------"
        echo "ERROR: ¡No puedes commitear con el modo de
producción activado en $CONFIG_FILE!"
        echo "Por favor, cambia '$PRODUCTION_FLAG' a 'False'
antes de commitear."
        echo
"------------------------------------------------------------"
        exit 1 # Aborta el commit
    fi
fi

echo "Pre-commit hook de config.py verificado. OK."
exit 0 # Permite el commit
```

4. Vuelve al directorio raíz de tu repositorio: `cd ..` (o `cd ../..` si estás más profundo).
5. **Prueba el hook (fallo):**
    - Abre `config.py` y **añade la línea `production_mode = True`** al final.
    - Intenta hacer un commit:

Unset

```
git add config.py
git commit -m "test: Intento de commit con modo produccion
activado"
# Deberías ver el ERROR y el commit debería ser abortado.
```

6. **Prueba el hook (éxito):**
    - Edita `config.py` y cambia `production_mode = True` a `production_mode = False` (o simplemente borra la línea si la añadiste solo para la prueba).
    - Intenta hacer un commit de nuevo:

Unset

```
git add config.py
git commit -m "fix: Desactivar modo produccion para commit"
# Ahora el commit debería pasar.
```

8. Posibles Errores y Soluciones

1. **Error: `No local changes to save` al hacer `git stash`.**
    - **Causa:** No tienes archivos modificados o en staging. `git stash` solo guarda cambios que Git ya detecta.
    - **Solución:** Modifica algunos archivos o crea nuevos y asegúrate de guardarlos. Si creas nuevos archivos, recuerda que para que `git stash` los guarde, necesitas usar `git stash -u` (incluir no rastreados) o `git stash -a` (incluir todos).
2. **Error: `fatal: bad object <hash>` al hacer `git cherry-pick`.**
    - **Causa:** El hash del commit que proporcionaste no existe en el repositorio o está mal escrito.
    - **Solución:** Usa `git log --oneline` o `git reflog` para encontrar el hash correcto del commit que deseas aplicar.
3. **Conflictos al aplicar un stash o al hacer `cherry-pick`.**
    - **Causa:** Los cambios que intentas aplicar (desde el stash o el cherry-pick) entran en conflicto con el contenido actual de tu Directorio de Trabajo o Área de Staging.
    - **Solución:** Git pausará la operación. Edita manualmente los archivos en conflicto, resuelve las marcas (`<<<<<<<`, `=======`, `>>>>>>>`), luego `git add <archivo_resuelto>` y finalmente `git stash pop` o `git cherry-pick --continue` para terminar. Si quieres abortar, usa `git stash pop --abort` o `git cherry-pick --abort`.
4. **Git Hook no se ejecuta o da error.**
    - **Causa:**
        - El archivo del hook no es ejecutable (`chmod +x`).
        - El nombre del archivo del hook es incorrecto (ej., `pre-commit.sh` en lugar de `pre-commit`).
        - El script del hook tiene errores de sintaxis o de lógica.
        - Estás intentando un commit con `--no-verify` (que salta los hooks).
    - **Solución:**
        - Asegúrate de ejecutar `chmod +x <nombre_del_hook>`.

        - Verifica que el nombre del archivo del hook es exactamente el nombre del evento de Git.
        - Depura el script del hook ejecutándolo directamente en la terminal.
        - Revisa la lógica del script para posibles errores.
5. **`git blame` muestra un historial que no esperabas.**
    - **Causa:** `git blame` muestra el último commit que tocó cada línea. Si una línea fue modificada recientemente, `blame` mostrará ese commit, no el commit original que la añadió. Además, los cambios de espacio en blanco o reformateo pueden "engañar" a `blame`.
    - **Solución:** Utiliza las opciones de `git blame` como `-w` (ignorar espacios en blanco), `-M` (detectar movimientos dentro del archivo), `-C` (detectar movimientos/copias desde otros archivos) para un análisis más profundo. Para encontrar el commit original que añadió una línea, `git log -S "<patrón>" --reverse` es a menudo más útil.

Conclusión de la Clase 8

¡Excelente trabajo, científicos de datos! Has desbloqueado una capa más profunda del poder de Git. Hoy hemos explorado herramientas avanzadas como **`git stash`** para una gestión flexible de cambios temporales, **`git tag`** para marcar hitos importantes en tu proyecto, y **`git cherry-pick`** para aplicar cambios específicos. Crucialmente, hemos introducido el concepto de **Git Hooks** para automatizar tareas de validación y análisis. Finalmente, has aprendido a usar `git blame` y `git bisect` para la **depuración y trazabilidad** en tu historial de código y datos.

Estás ganando un control significativo sobre tus repositorios y flujos de trabajo. En la próxima clase, profundizaremos en el **GitHub CLI** y otras herramientas que te permitirán interactuar con tus repositorios remotos de forma aún más eficiente.