Clase 9: Git Avanzado, Rebase y Herramientas CLI

Objetivos de la Clase

Al finalizar esta clase, serás capaz de:

- Comprender y aplicar el comando **`git rebase`** para reescribir y limpiar el historial de commits.
- Entender las **ventajas y desventajas de `rebase` vs. `merge`**.
- Utilizar **GitHub CLI (Command Line Interface)** para interactuar con tus repositorios remotos de forma eficiente desde la terminal.
- Explorar la utilidad de los **Submódulos Git** para gestionar dependencias de repositorios.
- Conocer y crear **alias en Git** para mejorar tu productividad en la línea de comandos.

1. `git rebase`: Reescribiendo el Historial de Commits

Mientras que `git merge` combina el historial de ramas creando un nuevo commit de fusión, `git rebase` te permite reescribir el historial, "moviendo" o "replicando" commits para que parezca que ocurrieron en un punto diferente en la línea de tiempo. El objetivo principal es mantener un historial de commits **más limpio y lineal**.

1.1. ¿Qué es `git rebase`?

`git rebase` toma una serie de commits de una rama y los "reaplica" sobre otra rama, como si hubieran sido escritos allí originalmente. En esencia, cambia el punto de partida (la base) de tu rama.

Sintaxis Básica:

```
Unset
git switch <tu_rama_de_trabajo> # Asegúrate de estar en la rama que quieres "rebasar"
git rebase <rama_base>          # Rebasea tu rama sobre la rama_base
```

- `tu_rama_de_trabajo`: La rama que contiene los commits que deseas reubicar.
- `rama_base`: La rama sobre la cual quieres "reaplicar" tus commits.

Ejemplo para Científicos de Datos: Imagina que estás trabajando en una rama `feat/algoritmo-nuevo` y la rama `main` ha avanzado con nuevas correcciones de datos. Quieres que tus cambios en `feat/algoritmo-nuevo` se vean como si hubieran sido hechos *después* de las últimas correcciones en `main`.

```
Unset
# Paso 1: Asegúrate de que tu rama 'main' local esté actualizada
git switch main
git pull origin main

# Paso 2: Cambia a tu rama de trabajo
git switch feat/algoritmo-nuevo

# Paso 3: Rebasea tu rama sobre 'main'
git rebase main
```

Git tomará los commits de `feat/algoritmo-nuevo` que no están en `main`, y los aplicará uno por uno sobre el último commit de `main`.

1.2. Resolución de Conflictos durante `rebase`

Durante un `rebase`, los conflictos son comunes, especialmente si tu rama y la rama base modificaron las mismas líneas. Git pausará el `rebase` y te lo indicará.

Proceso de Resolución:

1. Git te notificará el conflicto y el commit que lo causa.
2. Edita los archivos en conflicto manualmente (igual que con `merge`), elimina las marcas de conflicto (`<<<<<<<`, `=======`, `>>>>>>>`) y guarda los cambios.
3. Añade los archivos resueltos al área de staging: `git add <archivo_resuelto>`.
4. Continúa el `rebase`: `git rebase --continue`.
5. Si necesitas salir del `rebase` por completo, usa: `git rebase --abort`.

1.3. `git rebase -i` (Rebase Interactivo)

El rebase interactivo (`-i` o `--interactive`) es una herramienta poderosa que te permite **reescribir el historial de commits** de tu rama de formas muy específicas:

- **Reordenar** commits.
- **Editar** mensajes de commits.
- **`squash`** (combinar) múltiples commits en uno solo.
- **`fixup`** (combinar y descartar mensaje) commits.
- **`drop`** (eliminar) commits.
- **`edit`** (editar) commits para modificarlos o dividirlos.

Sintaxis:

```
Unset
git rebase -i HEAD~<N> # Para reescribir los últimos N commits desde HEAD
# O para reescribir commits desde un punto específico
git rebase -i <hash_del_commit_base_o_rama_base>
```

Esto abrirá tu editor de texto con una lista de los commits seleccionados y las opciones (`pick`, `reword`, `edit`, `squash`, `fixup`, `exec`, `drop`, etc.).

Ejemplo para Científicos de Datos: Tienes 3 commits en tu rama `feat/nuevo-analisis-eda` que representan:

1. "feat: Añadir visualización 1"
2. "chore: Corregir typo en comentario"
3. "feat: Añadir visualización 2" Quieres combinar las dos visualizaciones en un solo commit y eliminar el commit del typo.

```
Unset
git switch feat/nuevo-analisis-eda
git rebase -i HEAD~3 # O el hash del commit anterior a "Añadir visualización 1"
```

En el editor, cambiarías las líneas así:

```
Unset
pick <hash1> feat: Añadir visualización 1
squash <hash2> chore: Corregir typo en comentario  # Cambia
'pick' a 'squash' o 'fixup' para combinar
fixup <hash3> feat: Añadir visualización 2     # Cambia 'pick' a
'fixup' para combinar sin mantener el mensaje
```

Guarda y cierra el editor. Git te pedirá un nuevo mensaje para el commit combinado (si usaste `squash`).

1.4. Merge vs. Rebase: ¿Cuál usar?

Esta es una de las preguntas más debatidas en Git.

- **`git merge`**:
  - **Ventajas:** Es no destructivo; mantiene el historial original y explícito de todas las fusiones (`merge commits`). Es más fácil de entender para principiantes. Ideal para mantener un registro de cómo las ramas se unieron realmente.
  - **Desventajas:** Puede crear un historial de commits "sucio" con muchos `merge commits`, especialmente en proyectos con fusiones frecuentes.
  - **Cuándo usar:** Generalmente, para integrar cambios en **ramas públicas y compartidas** (ej., `main`, `develop`), donde no quieres reescribir el historial y es importante ver las fusiones explícitamente.
- **`git rebase`**:
  - **Ventajas:** Crea un historial de commits limpio y lineal, como si todos los cambios se hubieran hecho en una sola rama.
  - **Desventajas: ¡Reescribe el historial!** Esto puede ser muy peligroso en ramas que ya han sido publicadas y compartidas. Si reescribes el historial de una rama que otros ya tienen, causará problemas de sincronización para ellos (tendrían que hacer `git pull --rebase` o `git pull --force`).
  - **Cuándo usar:**
    - **En tus propias ramas de trabajo local, antes de publicarlas.** Para limpiar tu historial antes de abrir una Pull Request.
    - Para "rebasar" tu rama local sobre la `main` actualizada antes de abrir una PR, con el objetivo de evitar conflictos o limpiar el historial de tu PR.
    - **Nunca hagas rebase de una rama que ya ha sido publicada y que otros están utilizando.**

Regla de Oro: "Nunca hagas rebase en commits que ya has empujado a un repositorio público".

2. GitHub CLI (Command Line Interface)

GitHub CLI (`gh`) es una herramienta de línea de comandos que te permite interactuar directamente con GitHub (y sus funcionalidades de plataforma, no solo Git) desde tu terminal. Es una excelente manera de mejorar tu productividad sin tener que cambiar constantemente entre el navegador y la línea de comandos.

2.1. ¿Qué es GitHub CLI?

Mientras que Git (`git`) es el sistema de control de versiones, GitHub CLI (`gh`) es una **herramienta específica para la plataforma GitHub**. Te permite realizar tareas como crear Pull Requests, gestionar Issues, ver Notificaciones, crear Gists, y más, todo desde tu terminal.

2.2. Instalación y Configuración

- **Instalación:** `gh` no viene preinstalado. Debes instalarlo siguiendo las instrucciones para tu sistema operativo (ej., `brew install gh` en macOS, `apt install gh` en Linux).
- **Configuración Inicial:** Una vez instalado, debes autenticarte con tu cuenta de GitHub:

```
Unset

gh auth login
```

- Te guiará a través de un proceso interactivo para autenticarte vía navegador, configurar tu protocolo preferido (SSH recomendado) y subir tus claves SSH si es necesario.

2.3. Uso Práctico de `gh`

`gh` opera sobre el contexto del repositorio Git actual que está enlazado a GitHub.

- **`gh pr` (Pull Requests):**
  - `gh pr create`: Crear una nueva Pull Request interactivamente.
  - `gh pr list`: Listar todas las Pull Requests abiertas.
  - `gh pr checkout <número_pr>`: Descargar y cambiar a la rama de una PR para revisarla localmente. ¡Muy útil para científicos de datos que revisan código o notebooks de colegas!
  - `gh pr merge <número_pr>`: Fusionar una Pull Request.
  - `gh pr status`: Ver el estado de las PRs relevantes para ti.
- **`gh issue` (Issues):**

    - `gh issue create`: Crear una nueva Issue (ej., reportar un bug en un script de datos).
    - `gh issue list`: Listar Issues.
    - `gh issue view <número_issue>`: Ver detalles de una Issue.
- **`gh repo` (Repositorios):**
    - `gh repo create`: Crear un nuevo repositorio en GitHub.
    - `gh repo clone <usuario/repositorio>`: Clonar un repositorio de GitHub (alternativa a `git clone`).
    - `gh repo fork <usuario/repositorio>`: Crear un fork de un repositorio.
- **`gh gist` (Gists):**
    - `gh gist create <archivo>`: Crear un nuevo Gist (fragmento de código) desde un archivo.
    - `gh gist list`: Listar tus Gists.

Ejemplo para Científicos de Datos: Estás revisando un modelo y notas un comportamiento extraño. Quieres abrir un issue.

```
Unset
gh issue create --title "Bug: Inferencia del modelo devuelve NaNs" --body "Al pasar un DataFrame con nulos en columna X, el modelo produce valores NaN en la salida. Necesita un preprocesamiento adicional." --label "bug,data-quality" --assignee @me
```

Esto creará una issue directamente en tu repositorio de GitHub, asignándotela a ti mismo y etiquetándola para facilitar el seguimiento.

3. Trabajo con Submódulos

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.

3.1. ¿Para qué se usan los Submódulos en Ciencia de Datos?

- **Bibliotecas Compartidas / Módulos Internos:** Si tienes un conjunto de scripts de utilidad de Python comunes, modelos base pre-entrenados, o datasets de referencia que son gestionados como proyectos de Git independientes y que necesitas usar en varios de tus proyectos principales.

- **Componentes Reutilizables:** Un equipo puede tener un repositorio centralizado de "módulos de preprocesamiento" que cada proyecto de datos incorpora como un submódulo para asegurar la consistencia.
- **Separación de Responsabilidades:** Mantiene proyectos relacionados pero distintos separados, facilitando su gestión y desarrollo individual sin tener que copiar y pegar código.

3.2. Comandos Clave de Submódulos

- **Añadir un submódulo:**

```
Unset

git submodule add <URL_repositorio_submodulo>
<ruta_en_el_repo_principal>
# Ejemplo: git submodule add
https://github.com/mi-empresa/data-prep-utils.git
src/data_prep_utils
```

- Esto clonará el repositorio `data-prep-utils` en la ruta `src/data_prep_utils` dentro de tu repositorio principal y creará una entrada en el archivo `.gitmodules` (que también se versiona).
- **Clonar un repositorio con submódulos:** Si clonas un repositorio que tiene submódulos, estos no se clonarán automáticamente. Necesitas inicializarlos y actualizarlos:

```
Unset

git clone <URL_repo_principal>
cd <repo_principal>
git submodule update --init --recursive
```

    - `--init`: Inicializa los submódulos.
    - `--recursive`: Asegura que si un submódulo tiene submódulos anidados, también se inicialicen y actualicen.
- **Actualizar submódulos:** Cuando el submódulo tiene nuevos commits, necesitas actualizar la referencia en tu repo principal.

```
Unset

git submodule update --remote <ruta_del_submodulo>
```

- Después de esto, tu repositorio principal necesitará un nuevo commit para registrar que la referencia del submódulo ha avanzado.

Consideraciones:

- **Complejidad:** Los submódulos pueden añadir una capa de complejidad a tu flujo de trabajo de Git, especialmente en equipos grandes.
- **Flujo de Trabajo:** Requieren un flujo de trabajo específico: primero trabajas y commiteas en el submódulo, luego actualizas y commiteas la referencia en el repositorio principal.
- **Alternativas:** Para dependencias de código, a menudo se prefieren los gestores de paquetes (ej., `pip` en Python con `requirements.txt`). Para microservicios, a veces se usan monorepositorios sin submódulos, o herramientas como `repo`.

4. Alias en Git: Mejorando tu Productividad en la Terminal

Los alias en Git son atajos personalizados que puedes definir para comandos de Git más largos o para combinaciones de comandos que usas frecuentemente. Son una excelente forma de ahorrar tiempo y hacer tu vida en la terminal más cómoda.

4.1. ¿Qué son los Alias?

Son simples entradas en tu archivo de configuración de Git que mapean un nombre corto o una palabra clave a un comando o secuencia de comandos más complejos.

4.2. Crear Alias

Puedes crear alias a nivel global (para todos tus repositorios) o local (para un repositorio específico).

Sintaxis Global (recomendada para comandos de uso general):

```
Unset
git config --global alias.<nombre_del_alias>
"<comando_o_secuencia_de_comandos>"
```

Ejemplos de Alias Útiles para Científicos de Datos:

- **`git st` (para `git status --short`):** Para una vista rápida y concisa del estado.

```
Unset

git config --global alias.st "status -s"
```

- Uso: `git st`
- **`git co` (para `git checkout` o `git switch`):**

```
Unset

git config --global alias.co "switch" # Si prefieres 'switch'
# O para versiones antiguas de Git: git config --global alias.co
"checkout"
```

- Uso: `git co main`
- **`git ll` (para un `git log` conciso y gráfico):**

```
Unset

git config --global alias.ll "log --oneline --graph --all
--decorate"
```

- Uso: `git ll` (mostrará el historial de ramas de forma muy legible).
- **`git br` (para `git branch`):**

```
Unset

git config --global alias.br "branch"
```

- Uso: `git br`
- **`git cm` (para `git commit -m`):**

```
Unset

git config --global alias.cm "commit -m"
```

- Uso: `git cm "feat: Añadir nueva metrica"`
- **`git undo` (para `git reset --soft HEAD~1`):** Para deshacer el último commit localmente.

```
Unset

git config --global alias.undo "reset --soft HEAD~1"
```

- Uso: `git undo`
- **`git clean-all` (para limpiar directorio de trabajo y untracked):**

```
Unset

git config --global alias.clean-all '!git reset --hard HEAD &&
git clean -fd'
```

    - El `!` al inicio indica que el alias debe ejecutar un comando de shell (no un comando de Git interno).
    - `reset --hard HEAD`: Descarta todos los cambios locales.
    - `clean -fd`: Elimina archivos y directorios no rastreados.

4.3. Listar Tus Alias

```
Unset
git config --global --get-regexp ^alias\.
# O si creaste un alias para esto mismo:
# git config --global alias.aliases "config --global --get-regexp
^alias\."
# Uso: git aliases
```

5. Actividad Práctica (Laboratorio 9)

Vamos a practicar `git rebase` y a explorar el uso de GitHub CLI y alias.

Paso 1: Preparación del Entorno

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

```
Unset

mkdir clase9-lab-ds
cd clase9-lab-ds
git init
```

2. Crea un archivo `data_script.py` con contenido inicial:

```
Unset

# data_script.py
import pandas as pd

def load_data(path):
    return pd.read_csv(path)

def preprocess_data(df):
    return df.dropna()
```

3. Haz el primer commit:

```
Unset

git add data_script.py
git commit -m "feat: Initial data script with load and
preprocess"
```

Paso 2: Práctica con `git rebase` (No interactivo)

1. **Crea una rama `feat/nuevo-preprocesamiento` y cámbiate a ella:**

```
Unset

git switch -c feat/nuevo-preprocesamiento
```

2. En esta rama, haz dos commits:
    - **Commit 1:** Modifica `data_script.py` para añadir una función de normalización.

```
Unset

# data_script.py (modificado)
import pandas as pd
from sklearn.preprocessing import MinMaxScaler

def load_data(path):
    return pd.read_csv(path)

def preprocess_data(df):
    return df.dropna()

def normalize_data(df, columns):
    scaler = MinMaxScaler()
    df[columns] = scaler.fit_transform(df[columns])
    return df
```bash
git add data_script.py
git commit -m "feat: Add data normalization function"
```

    - **Commit 2:** Modifica `data_script.py` para añadir una función de codificación.

```
Unset

# data_script.py (modificado)
import pandas as pd
```
```

```
from sklearn.preprocessing import MinMaxScaler, OneHotEncoder

def load_data(path):
    return pd.read_csv(path)

def preprocess_data(df):
    return df.dropna()

def normalize_data(df, columns):
    scaler = MinMaxScaler()
    df[columns] = scaler.fit_transform(df[columns])
    return df

def encode_categorical(df, columns):
    encoder = OneHotEncoder()
    # Simplificado, solo para ejemplo
    encoded_data = encoder.fit_transform(df[columns]).toarray()
    return pd.concat([df.drop(columns=columns),
pd.DataFrame(encoded_data)], axis=1)
```bash
git add data_script.py
git commit -m "feat: Implement one-hot encoding for categorical
features"
```

3. Vuelve a la rama `main`:

```
Unset

git switch main
```

4. En `main`, haz un nuevo commit que simule un cambio en el "pasado" (ej., una corrección de un typo):
```

Unset

```
# data_script.py (modificado en main)
import pandas as pd

def load_data(path):
    return pd.read_csv(path)

def preprocess_data(df):
    # Corregir un typo en un comentario
    # Esto limpia los datos de valores nulos
    return df.dropna()
```

Unset

```
git add data_script.py
git commit -m "fix: Correct typo in preprocess_data comment"
```

5. Visualiza el historial con `git log --oneline --graph --all`. Verás que las ramas `main` y `feat/nuevo-preprocesamiento` han divergido.
6. **Ahora, rebasea `feat/nuevo-preprocesamiento` sobre `main`:**

Unset

```
git switch feat/nuevo-preprocesamiento
git rebase main
# Si hay conflictos, resuélvelos, git add, y git rebase
--continue
```

7. Visualiza el historial con `git log --oneline --graph --all`. Observa cómo los commits de `feat/nuevo-preprocesamiento` ahora aparecen *después* del commit de `main`.

Paso 3: Práctica con `git rebase -i` (Interactivo)

1. Asegúrate de estar en `feat/nuevo-preprocesamiento`.
2. Haz un par de commits "desordenados" o "pequeños" en esta rama:
    - **Commit A:** `echo "temp_line_a" >> data_script.py` -> `git commit -m "temp: temp commit a"`
    - **Commit B:** `echo "temp_line_b" >> data_script.py` -> `git commit -m "temp: temp commit b"`
    - **Commit C:** `echo "temp_line_c" >> data_script.py` -> `git commit -m "temp: temp commit c"`
3. Ahora, inicia un rebase interactivo para los últimos 3 commits:

```
Unset

git rebase -i HEAD~3
```

4. En el editor:
    - Cambia `pick` a `squash` para el "temp commit b" (combinarlo con A).
    - Cambia `pick` a `fixup` para el "temp commit c" (combinarlo con A sin mantener su mensaje).
    - Cambia el mensaje del primer commit si es necesario.
5. Guarda y cierra el editor. Si usaste `squash`, te pedirá un nuevo mensaje para el commit combinado.
6. Verifica `git log --oneline`. Deberías ver que los tres commits se han combinado en uno solo.

Paso 4: Exploración de GitHub CLI (`gh`) (Requiere GitHub CLI instalado y autenticado)

1. Asegúrate de que estás en un repositorio que tienes enlazado a GitHub (puedes usar el del laboratorio de la Clase 5).
2. **Lista tus Pull Requests:**

```
Unset

gh pr list
```

3. **Lista tus Issues:**

```
Unset

gh issue list
```

4. **Crea una Issue de prueba:**

```
Unset

gh issue create --title "Investigar baja precisión en modelo
v1.0" --body "El modelo de clasificación de churn V1.0 tiene una
precisión del 75%, cuando se esperaba 80%. Investigar posibles
causas: datos de entrada, hiperparámetros, etc." --label
"bug,investigation"
```

5. *Verifica en GitHub si se creó la issue.*
6. **Crea un `gist` de un snippet de código:**
    - Crea un archivo `snippet.py` con una función simple:

```
Unset

# snippet.py
def calculate_rmse(y_true, y_pred):
    return ((y_true - y_pred) ** 2).mean() ** 0.5
```

    - Crea el `gist`:

```
Unset

gh gist create snippet.py -d "Función de cálculo de RMSE para
modelos de regresión"
```

    - *El comando te devolverá la URL del gist creado.*

Paso 5: Creación de Alias en Git

1. Abre tu terminal y crea algunos alias útiles:

Unset

```
git config --global alias.st "status -s"
git config --global alias.ll "log --oneline --graph --all
--decorate"
git config --global alias.br "branch"
git config --global alias.co "switch"
git config --global alias.undo "reset --soft HEAD~1"
```

2. Verifica que los alias funcionan:

Unset

```
git st
git ll
git br
git co main
git undo # ¡Solo si tienes un commit que puedas deshacer!
```

3. Lista todos tus alias:

Unset

```
git config --global --get-regexp ^alias\.
```

6. Posibles Errores y Soluciones

1. **Error: `fatal: It seems that a rebase process is already in progress...`**
    - **Causa:** Intentaste iniciar un nuevo `rebase` mientras uno anterior no se completó (o se abortó incorrectamente).
    - **Solución:** Primero, asegúrate de que no estás en medio de un `rebase` (ver `git status`). Si lo estás, puedes `git rebase --continue` si ya resolviste conflictos, o `git rebase --abort` para descartar el rebase en curso.
2. **Conflictos repetitivos durante `git rebase --continue`.**

    - **Causa:** Tienes muchos commits en tu rama que colisionan con la rama base, o las resoluciones no son correctas.
    - **Solución:** A veces, es más fácil abortar el `rebase` (`git rebase --abort`), hacer un `merge` simple de la rama base a tu rama (`git merge <rama_base>`) para resolver todos los conflictos de una vez, y luego reintentar el `rebase` si el historial lineal es crítico. Asegúrate de que tus commits sean pequeños y atómicos para minimizar conflictos.
3. **Error: `You are trying to rebase a branch that is already a descendant of '<rama_base>'`**
    - **Causa:** Intentaste rebasar tu rama sobre una rama que ya es un ancestro directo. Es decir, tu rama ya contiene todos los commits de la rama base, y no hay nada que rebasar.
    - **Solución:** Esto no es un error grave, solo una notificación. Significa que tu rama ya está al día. Puedes hacer `git pull <rama_base>` en lugar de `rebase` en estos casos, o simplemente ya no necesitas rebasar.
4. **Comandos de `gh` fallan con "authentication required" o "no git remotes found".**
    - **Causa:** GitHub CLI no está autenticado, o el repositorio local no está enlazado a un repositorio remoto en GitHub.
    - **Solución:** Ejecuta `gh auth login` para autenticarte. Asegúrate de que tu directorio de trabajo es un repositorio Git y está enlazado a GitHub (`git remote -v`).
5. **Alias no funcionan o dan error.**
    - **Causa:** Error de sintaxis en la definición del alias, o el comando al que apunta el alias no existe. El `!` en los alias indica un comando de shell.
    - **Solución:** Revisa la sintaxis de tu `git config --global alias.<nombre>` cuidadosamente. Depura el comando original antes de ponerlo en un alias.

Conclusión de la Clase 9

¡Excelente trabajo, científicos de datos! Has desbloqueado algunas de las herramientas más poderosas y complejas de Git. Hoy hemos dominado **`git rebase`**, una técnica fundamental para reescribir y limpiar tu historial de commits, entendiendo sus implicaciones y cuándo usarlo de forma segura. Hemos explorado la eficiencia de **GitHub CLI**, que te permite interactuar con tu plataforma remota directamente desde la terminal, y la utilidad de los **alias en Git** para optimizar tu productividad. Finalmente, hemos tocado los **submódulos** como una forma de gestionar dependencias de repositorios.

Estás equipado con un arsenal avanzado de herramientas Git que te harán un colaborador mucho más eficiente y un gestor de proyectos de datos más metódico. En nuestra próxima y última clase, integraremos todos estos conocimientos en un **proyecto final colaborativo**, donde pondrás a prueba todo lo aprendido.