# Clase 3: Viajando en el Tiempo y Ramas Locales para Científicos de Datos

## Objetivos de la Clase

Al finalizar esta clase, serás capaz de:

- Comprender el concepto de **HEAD** y cómo Git lo utiliza para navegar por el historial.
- Aprender a **deshacer cambios en el historial local** de forma segura y controlada.
- Dominar la **creación y gestión de ramas** en Git.
- Entender la **importancia estratégica de las ramas** para la experimentación y el desarrollo paralelo en proyectos de ciencia de datos.

## 1. El Puntero `HEAD`: Tu Posición en el Historial de Commits

En Git, `HEAD` es un concepto fundamental. Es un **puntero** que siempre apunta al **último commit de la rama en la que te encuentras actualmente**. En términos más simples, `HEAD` te indica dónde estás "parado" en el historial de tu repositorio.

Image of Git HEAD Pointer

### 1.1. ¿Por qué es importante `HEAD`?

- **Contexto Actual:** Siempre sabes a qué commit y a qué rama se refiere tu "estado actual".
- **Navegación:** Muchos comandos de Git (`checkout`, `reset`, `revert`) usan `HEAD` como referencia para moverse o modificar el historial.
- **Identificación:** Como los commits se identifican con hashes largos (ej., `e137e9b`), `HEAD` proporciona una referencia legible y dinámica a tu ubicación.

### 1.2. ¿Cómo saber dónde apunta `HEAD`?

Puedes ver a dónde apunta `HEAD` con los siguientes comandos:

- `git symbolic-ref HEAD`: Muestra la referencia simbólica (normalmente, la rama actual).

```
Unset
git symbolic-ref HEAD

# Salida: refs/heads/main
```

- Esto significa que `HEAD` está apuntando a la rama `main`.
- `git rev-parse HEAD`: Muestra el hash SHA-1 completo del commit al que apunta `HEAD`.

```
Unset
git rev-parse HEAD

# Salida: a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6q7r8s9t0
```

- Este es el identificador único del commit actual.

**Analogía para Científicos de Datos:** Piensa en `HEAD` como el "punto de control" o "estado del experimento" actualmente cargado en tu entorno de trabajo. Cuando haces un `commit`, `HEAD` avanza a esa nueva "fotografía". Si cambias a otra rama de experimentación, `HEAD` se mueve con ella.

## 2. Deshaciendo Cambios Localmente (Sin Publicar)

Una de las grandes ventajas de Git es su capacidad de "deshacer" errores. Sin embargo, la forma de deshacer depende de si los cambios ya han sido publicados en un repositorio remoto (lo veremos más adelante) o si solo existen en tu **repositorio local**. En esta clase, nos enfocaremos en deshacer cambios que aún no has compartido con nadie.

### 2.1. Deshacer el Último Commit (`git reset`)

El comando `git reset` te permite mover `HEAD` (y opcionalmente tu rama) a un commit anterior, deshaciendo commits recientes. Es una herramienta potente y debe usarse con precaución, especialmente si los commits ya han sido compartidos.

**Sintaxis y Tipos de `reset`:**

- `git reset --soft HEAD~1` (Mantener cambios en Staging):
  - **¿Qué hace?** Deshace el último commit, pero **mantiene los cambios de ese commit en tu Área de Staging**. El `HEAD` de tu rama se mueve un commit hacia atrás. Es útil si quieres volver a hacer el mismo commit con un mensaje diferente o añadir más cambios antes de volver a commitear.
  - `HEAD~1` se refiere al commit inmediatamente anterior a `HEAD`. Puedes usar `HEAD~N` para retroceder `N` commits.
- `git reset --mixed HEAD~1` (Mantener cambios en Working Directory):
  - **¿Qué hace?** Deshace el último commit, saca los cambios del Área de Staging, pero los **mantiene en tu Directorio de Trabajo como cambios**

**modificados**. Es el comportamiento por defecto de `git reset` si no especificas `--soft` o `--hard`.

- `git reset --hard HEAD~1` (Eliminar cambios completamente):
    - **¿Qué hace?** Deshace el último commit y **elimina completamente todos los cambios** de ese commit tanto del Área de Staging como del Directorio de Trabajo. **¡Estos cambios se pierden!** Úsalo solo si estás absolutamente seguro de que quieres descartar todo el trabajo.
    - **¡ADVERTENCIA!** Este es el comando más destructivo. No hay "papelera de reciclaje" para los cambios descartados con `--hard`.

**Ejemplo para Científicos de Datos (simulando un error en el último commit):**

Imagina que hiciste un commit con un ajuste de hiperparámetros, pero te das cuenta de que lo hiciste en la rama incorrecta, o el mensaje es erróneo y quieres añadir más cosas.

Unset

```
# Paso 1: Simula un commit erróneo en el último momento
# Asegúrate de tener cambios commiteados
# Puedes añadir un archivo si tu repositorio está vacío, o modificar
uno existente:
# echo "import math" > temp_script.py
# git add temp_script.py
# git commit -m "feat: Añadir script temporal"

# Luego, simula el commit que quieres deshacer
echo "param = 0.01" >> temp_script.py
git add temp_script.py
git commit -m "feat: Ajuste inicial de learning rate"
# Output: [main abcdef1] feat: Ajuste inicial de learning rate

# Paso 2: Deshacer el último commit, manteniendo los cambios en
staging
git reset --soft HEAD~1
# Salida: HEAD is now at <anterior_hash> <mensaje_anterior_commit>
# Verifica el estado:
git status
# Salida: Changes to be committed: modified: temp_script.py
```

```
# Los cambios del commit deshecho están de nuevo en el staging.
# Puedes editar el archivo, añadir más cosas, o simplemente hacer un
nuevo commit.

# Paso 3: Opcional - deshacer completamente (solo si estás seguro)
# Si quisieras borrar los cambios del script temp_script.py por
completo, usarías:
# git reset --hard HEAD~1
# ¡CUIDADO! Esto borraría el contenido de temp_script.py a como
estaba antes del commit.
```

### 2.2. Arreglar el Último Commit (`git commit --amend`)

Si el último commit que hiciste tiene un mensaje incorrecto, o te olvidaste de incluir un pequeño cambio (o un archivo) que pertenece lógicamente a ese commit, puedes "enmendarlo" en lugar de crear uno nuevo.

**Sintaxis:**

- **Cambiar solo el mensaje del último commit:**

```
Unset
git commit --amend -m "Este es el mensaje correcto ahora"
```

- **Añadir cambios al último commit:**
    1. Realiza tus modificaciones en los archivos.
    2. Añade los archivos modificados al Área de Staging: `git add <archivo1> <archivo2>`.
    3. Ejecuta el comando para enmendar el commit:

```
Unset
git commit --amend --no-edit # Omitir si quieres editar el mensaje
```

    4. Si omites `--no-edit`, se abrirá el editor de texto con el mensaje anterior para que lo edites.

**Importante:** `git commit --amend` no crea un nuevo commit. **Reemplaza el último commit con uno nuevo**, manteniendo el mismo "padre" pero con un nuevo hash. Por esta razón, **nunca uses** `--amend` **en un commit que ya haya sido publicado** en un repositorio remoto y compartido con otros, ya que reescribiría el historial y causaría problemas de sincronización. Para commits publicados, usarás `git revert` (Clase 5).

Ejemplo para Científicos de Datos:

Imagina que commiteaste un script de visualización, pero olvidaste incluir una pequeña función de ayuda o una librería en los imports.

Unset

```
# Simula un commit previo (si tu repo está vacío)
# echo "import matplotlib.pyplot as plt" > visualize_data.py
# git add visualize_data.py
# git commit -m "feat: Script inicial de visualización"

# Paso 1: Te das cuenta de que olvidaste una línea en
visualize_data.py
echo "plt.style.use('ggplot')" >> visualize_data.py # Añade una
línea
echo "import seaborn as sns" >> visualize_data.py # O añades un
import
git status # Verás visualize_data.py como Modified

# Paso 2: Añade los cambios al staging
git add visualize_data.py

# Paso 3: Enmienda el último commit para incluir estos cambios
git commit --amend --no-edit
# Salida: [main fedcba9] feat: Script inicial de visualización
# Esto ha reemplazado el commit anterior con uno nuevo que incluye
los cambios.
# El hash del commit habrá cambiado.
```

## 3. Ramas en Git: Concepto y Utilidad para Científicos de Datos

Las ramas son el pilar del desarrollo colaborativo y la experimentación en Git. Una **rama** es una línea de desarrollo independiente que diverge de la rama principal (o de otra rama).

Image of Un poco de teoría - Qué es una rama

### 3.1. ¿Qué son las ramas?

Conceptualmente, una rama es solo un puntero movible a un commit. Cuando creas una nueva rama, Git simplemente crea un nuevo puntero que apunta al mismo commit que tu rama actual. Cuando haces commits en esa nueva rama, el puntero de la rama avanza, dejando a la rama original intacta.

### 3.2. ¿Para qué se utilizan las ramas en Ciencia de Datos?

- **Aislamiento de Experimentos:** Es la forma más segura de probar nuevas hipótesis, algoritmos, preprocesamientos o visualizaciones. Puedes crear una rama `exp-modelo-X` para probar un modelo de regresión, mientras la rama `main` mantiene la versión de producción.
- **Desarrollo de Nuevas Features:** Implementar una nueva función de `feature engineering` o un nuevo módulo de evaluación en una rama `feat-nueva-metrica` sin afectar el flujo de trabajo principal.
- **Reproducción y Solución de Bugs:** Si un colega reporta un bug en un script de análisis que está en `main`, puedes crear una rama `bugfix-error-filtrado` para aislar el problema, corregirlo y probarlo sin contaminar la rama principal.
- **Colaboración en Equipo:** En un equipo de científicos de datos, cada persona puede trabajar en su propia rama en diferentes partes del proyecto (ej. ingesta de datos, limpieza, entrenamiento de modelo, evaluación, despliegue) simultáneamente.

### 3.3. Rama Principal (`main` o `master`)

Históricamente, la rama principal se llamaba `master`. Sin embargo, la convención moderna y recomendada (promovida por plataformas como GitHub y GitLab) es usar `main`. Esta rama representa la versión estable y funcional de tu proyecto.

## 4. Trabajando con Ramas: Comandos Clave

Dominar la gestión de ramas es esencial para cualquier proyecto que utilice Git.

### 4.1. Crear una Nueva Rama (`git branch`)

El comando `git branch` se usa para crear nuevas ramas sin cambiarte a ellas.

- **Sintaxis:**

```
Unset
git branch <nombre_de_la_rama>
```

- La nueva rama se creará a partir del commit al que apunta `HEAD` actualmente.

**Ejemplo:**

```
Unset

# Crea una rama para un nuevo experimento
git branch experimento-a-b
```

### 4.2. Cambiar a una Rama Existente (`git switch`)

El comando `git switch` (introducido en Git 2.23 como una alternativa más limpia a `git checkout` para cambiar de rama) se usa para cambiar entre ramas.

- **Sintaxis:**

```
Unset
git switch <nombre_de_la_rama>
```

- Esto actualizará tus archivos en el Directorio de Trabajo para que coincidan con el estado de la rama a la que te cambias.

**Ejemplo:**

```
Unset
# Cambia a la rama de tu experimento
git switch experimento-a-b
# Salida: Switched to branch 'experimento-a-b'
```

### 4.3. Crear y Cambiar a una Nueva Rama en un Solo Paso (`git switch -c`)

Esta es una forma muy común de crear una rama y empezar a trabajar en ella inmediatamente.

- **Sintaxis:**

```
Unset
git switch -c <nombre_de_la_nueva_rama>
```

- Esto es equivalente a `git branch <nombre_de_la_nueva_rama>` seguido de `git switch <nombre_de_la_nueva_rama>`.

**Ejemplo:**

```
Unset

# Crea una rama para desarrollar un nuevo módulo de preprocesamiento y cámbiate a ella
git switch -c feat-preprocesamiento-avanzado
# Salida: Switched to a new branch 'feat-preprocesamiento-avanzado'
```

### 4.4. Listar Ramas (`git branch`)

Puedes ver qué ramas existen en tu repositorio local y en cuál te encuentras actualmente.

- **Sintaxis:**

```
Unset
git branch
```

- La rama actual estará marcada con un asterisco (`*`).

**Ejemplo:**

```
Unset

git branch
# Salida:
#   experimento-a-b
# * feat-preprocesamiento-avanzado
#   main
```

Esto indica que estás en la rama `feat-preprocesamiento-avanzado`.

- `git branch --show-current`: Muestra solo el nombre de la rama actual.

```
Unset
git branch --show-current

# Salida: feat-preprocesamiento-avanzado
```

**4.5. Visualizar el Historial de Ramas Gráficamente (`git log --graph`)**

Para entender realmente cómo las ramas se bifurcan y se unen, la vista gráfica del historial es invaluable.

- **Sintaxis:**

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

  - `--oneline`: Cada commit en una línea.
  - `--graph`: Dibuja un grafo ASCII.
  - `--all`: Muestra todas las ramas (locales y remotas).

**Ejemplo de Salida (Conceptual):**

```
Unset

* fe12a3b (HEAD -> feat/new-model) feat: Implementación final de
RandomForest
* a7b8c9d feat: Añadir cross-validation al modelo
| * d1e2f3g (main) refactor: Mejorar loggin de datos
|/
* 1234567 Initial project setup
```

## 5. Actividad Práctica (Laboratorio 3)

Continuaremos trabajando en un repositorio local para aplicar los conceptos de `HEAD` y ramificación.

**Paso 1: Preparación del Entorno**

1. Si no lo hiciste en la Clase 2, crea un nuevo directorio para esta clase y entra en él:

```
Unset
mkdir clase3-lab-ds
cd clase3-lab-ds
```

2. Inicializa un nuevo repositorio Git:

```
Unset
git init
```

3. Crea un archivo `main_script.py` y añádele contenido básico de Python:

```
Unset
# main_script.py
print("Este es el script principal del proyecto.")
print("Version 1.0 de la funcionalidad.")
```

4. Realiza el primer commit:

```
Unset
git add main_script.py

git commit -m "feat: Initial version of main script"
```

5. Confirma con `git status` y `git log --oneline`.

**Paso 2: Práctica con `HEAD` y Deshaciendo Commits (Localmente)**

1. Verifica dónde apunta `HEAD`:

```
Unset
git symbolic-ref HEAD

git rev-parse HEAD
```

2. Ahora, haz una modificación en `main_script.py` y un nuevo commit:

```
Unset
# main_script.py (añade una línea)

print("Este es el script principal del proyecto.")
print("Version 1.0 de la funcionalidad.")
print("Añadida una nueva característica de logging.")
```bash
git add main_script.py
git commit -m "feat: Add logging feature"
```

3. Verifica el historial con `git log --oneline`. Ahora tienes dos commits.
4. **Simula un error:** Te das cuenta de que el último commit ("Add logging feature") debería haber sido parte de otro cambio, o el mensaje es incorrecto y quieres añadir más cosas.
    - **Deshaz el último commit, manteniendo los cambios en staging:**

```
Unset
git reset --soft HEAD~1
```

    - Verifica `git status`. Deberías ver `main_script.py` en "Changes to be committed".
    - Verifica `git log --oneline`. El último commit debería haber desaparecido del historial.
    - Ahora, modifica `main_script.py` de nuevo (ej. cambia la línea de logging o añade otra).

```
Unset
# main_script.py (modifica la línea de logging)


print("Este es el script principal del proyecto.")
print("Version 1.0 de la funcionalidad.")
print("Añadida una característica de logging mejorada.")
```

        - 
```

- Añade `main_script.py` al staging:

```
Unset
git add main_script.py
```

- **Enmienda el commit anterior** (el original "Initial version of main script") con estos nuevos cambios y un mensaje actualizado:

```
Unset
git commit --amend -m "feat: Add and improve initial logging feature"
```

- Verifica `git log --oneline`. Ahora solo deberías ver un commit, pero con el mensaje y el contenido actualizado.

**Paso 3: Creación y Navegación de Ramas**

1. Asegúrate de estar en la rama `main` y que el directorio de trabajo esté limpio (`git status`).
2. **Crea una nueva rama para un experimento de modelo:**

```
Unset
git switch -c exp-modelado-decision-tree
```

3. Verifica que te has cambiado de rama con `git branch --show-current`.
4. En esta nueva rama, crea un script `model_decision_tree.py` y añade contenido simple:

```
Unset


# model_decision_tree.py
from sklearn.tree import DecisionTreeClassifier
```

```
def train_decision_tree(X, y):
    model = DecisionTreeClassifier()
    model.fit(X, y)
    print("Decision Tree model trained.")
    return model
```

5. Realiza un commit en esta rama:

```
Unset
git add model_decision_tree.py

git commit -m "feat: Implement Decision Tree model for experiment"
```

6. Ahora, **vuelve a la rama** `main`:

```
Unset
git switch main
```

    - **Observación:** Verifica el contenido de tu directorio. El archivo `model_decision_tree.py` debería haber desaparecido (o no ser visible), ya que no existe en la rama `main`. Esto demuestra el aislamiento de las ramas.
    - Verifica `git status`. Debería estar limpio.
7. En la rama `main`, realiza un cambio independiente. Modifica `main_script.py` para añadir una función de utilidad:

```
Unset
# main_script.py (añade una función de utilidad)
print("Este es el script principal del proyecto.")
print("Version 1.0 de la funcionalidad.")
print("Añadida una característica de logging mejorada.")

def utility_function():
```

```
    print("Función de utilidad principal.")
```bash
git add main_script.py
git commit -m "refactor: Add a general utility function"
```

**Paso 4: Visualizar la Divergencia de Ramas**

1. Mira el historial completo de todas las ramas en un formato gráfico:

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

2. Deberías ver cómo la rama `exp-modelado-decision-tree` se bifurca de `main` y cómo ambas tienen commits independientes después de la bifurcación.

## 6. Posibles Errores y Soluciones

Aquí te presento algunos errores comunes que podrías encontrar en esta clase y cómo solucionarlos.

1. **Error:** `fatal: could not reset to 'HEAD~1'`
    - **Causa:** Esto ocurre si intentas `git reset HEAD~1` y estás en el primer commit del repositorio (no hay `HEAD~1` al que retroceder).
    - **Solución:** Asegúrate de tener al menos dos commits en tu rama antes de intentar este comando. Si estás en el primer commit y quieres eliminarlo, tendrías que recrear el repositorio (borrar `.git` y `git init` de nuevo), pero esto es raro.
2. **Error:** `fatal: A branch named 'X' already exists.` **al usar** `git switch -c`
    - **Causa:** Intentaste crear una nueva rama con `git switch -c <nombre_rama>`, pero una rama con ese nombre ya existe.
    - **Solución:** Si quieres cambiar a la rama existente, usa `git switch <nombre_rama>`. Si quieres crear una nueva rama, elige un nombre diferente.
3. **Error:** `Your local changes to the following files would be overwritten by checkout:` **al cambiar de rama.**
    - **Causa:** Tienes cambios sin commitear (modificados o en staging) en tu Directorio de Trabajo, y al cambiar a otra rama, esos cambios entrarían en
```

conflicto o serían sobrescritos por los archivos de la nueva rama. Git te protege de perder tu trabajo.
    - **Solución:**
        - **Opción A (Recomendada):** Haz un `commit` de tus cambios en la rama actual antes de cambiar.
        - **Opción B:** Si los cambios son temporales y no quieres commitearlos, guárdalos con `git stash` (lo veremos en la Clase 8).
        - **Opción C (¡Solo si estás seguro de descartar!):** Deshaz tus cambios con `git restore .` antes de cambiar de rama.
4. **Error:** `fatal: You are on a branch whose name is too long or contains invalid characters.`
    - **Causa:** Los nombres de rama tienen ciertas restricciones. No pueden contener espacios, ni caracteres especiales (como `~`, `^`, `:`, `?`, `*`), no pueden terminar en `.lock`, no pueden empezar con `.` o `@`, etc. Es mejor usar guiones (`-`) o barras (`/`) para separar palabras.
    - **Solución:** Elige un nombre de rama más corto y descriptivo, usando solo letras minúsculas, números, guiones (`-`) y barras (`/`).

## Conclusión de la Clase 3

¡Excelente trabajo, científicos de datos! Hoy hemos dado un paso crucial al entender el concepto de `HEAD` y cómo manipular el historial de commits en nuestro repositorio local. Lo más importante, hemos aprendido el poder de las **ramas**: cómo crearlas, cómo navegar entre ellas y por qué son tan vitales para la experimentación, el desarrollo paralelo y la colaboración en proyectos de ciencia de datos.

Estás construyendo las habilidades necesarias para trabajar de manera eficiente y segura con Git. En la próxima clase, llevaremos nuestro aprendizaje al siguiente nivel al integrar nuestros repositorios locales con **repositorios remotos** y comenzar a colaborar.