Clase 1: Introducción a Git y Control de Versiones para Científicos de Datos

Objetivos de la Clase

Al finalizar esta clase, serás capaz de:

- Comprender la **necesidad y las ventajas** de utilizar un sistema de control de versiones en el ámbito de la ciencia de datos.
- Conocer la **historia y los conceptos fundamentales** que sustentan a Git.
- Realizar la **instalación y configuración inicial** de Git en tu sistema operativo.
- Familiarizarte con los **comandos básicos de la terminal** necesarios para interactuar con Git.

1. ¿Qué es el Control de Versiones y Por Qué es Importante en Ciencia de Datos?

1.1. Definición

Un **Sistema de Control de Versiones (VCS - Version Control System)** es una herramienta que registra cada cambio que se realiza en el código (scripts de Python, notebooks Jupyter), archivos de configuración o incluso documentación de un proyecto a lo largo del tiempo. Esto permite a los científicos de datos llevar un seguimiento detallado de cada modificación, saber quién la hizo, cuándo y por qué.

Imagina que estás desarrollando un modelo predictivo, y cada vez que ajustas un hiperparámetro, cambias una variable de entrada o actualizas el script de preprocesamiento, guardas una "copia" con una nota sobre lo que modificaste. Si más tarde descubres que una modificación empeoró el rendimiento del modelo, puedes volver a una versión anterior y estable. Un VCS hace esto de forma automatizada y mucho más potente para todos tus recursos de datos.

1.2. Ventajas Clave para Científicos de Datos

La adopción de un VCS, especialmente Git, ofrece beneficios fundamentales en proyectos de ciencia de datos:

- **Reproducibilidad:** Asegura que puedes recrear exactamente cualquier resultado o modelo en cualquier momento, al tener un registro preciso de los scripts, dependencias (si se gestionan) y configuraciones utilizados. Esto es vital para la investigación y el despliegue.
- **Colaboración Eficaz:** Múltiples científicos de datos pueden trabajar en el mismo proyecto (diferentes scripts de limpieza, módulos de modelos, o análisis exploratorios) simultáneamente sin sobrescribir el trabajo de los demás. Git gestiona la integración de sus cambios.
- **Recuperación Sencilla:** ¿Has introducido un error en tu script de limpieza de datos que corrompe tu dataset? ¿Necesitas volver a una versión anterior y funcional de tu modelo? Con un VCS, es tan sencillo como retroceder a un punto en el tiempo.
- **Experimentación Segura:** Puedes crear "ramas" (bifurcaciones) de tu código y notebooks para probar nuevas hipótesis, algoritmos o enfoques de análisis sin afectar la versión principal y estable de tu proyecto o modelo en producción. Una vez terminados y probados, estos cambios se pueden integrar.
- **Auditoría y Trazabilidad:** Mantén un historial claro de cómo evolucionaron tus scripts, modelos y resultados, quién hizo qué cambio, y cuándo. Esto es fundamental para la depuración de errores, la colaboración y para cumplir con requisitos de auditoría.

1.3. Analogía: El Laboratorio del Científico de Datos

Piensa en un sistema de control de versiones no solo como una línea de tiempo lineal, sino como un **laboratorio de alta tecnología** para tus proyectos de datos.

- El **tronco principal** (tu rama `main` o `master`) sería el código y los modelos más estables y listos para ser utilizados o desplegados.
- Las **ramas** son como los experimentos o proyectos paralelos que inicias en tu laboratorio: te permiten desviarte para desarrollar un nuevo algoritmo, probar una técnica de visualización o limpiar una parte específica de un dataset, sin contaminar tu entorno principal.
- Una vez que tu experimento (rama) está listo, validado y sus resultados son concluyentes, puedes **fusionarlo** de nuevo en tu trabajo principal.

2. Problemas Comunes que Soluciona el Control de Versiones

Antes de los VCS, los científicos de datos (y desarrolladores en general) se enfrentaban a desafíos significativos que hoy son impensables:

- **"Versión Final_final_V2.ipynb"**: Gestión manual de versiones de scripts y notebooks, lo que lleva a la confusión, duplicación de archivos y la pérdida de cambios.
- **Sobrescritura Accidental:** Trabajar en el mismo script de Python o archivo de datos con un colega sin un sistema de coordinación resultaba en que uno sobrescribía el trabajo del otro.
- **"Funcionó ayer, hoy no"**: Dificultad para rastrear regresiones o errores que surgen después de cambios, sin saber qué modificación específica los causó.
- **Imposibilidad de Replicar un Experimento:** La falta de un registro claro de los scripts, dependencias y datasets utilizados en un experimento dificulta la reproducibilidad por parte de otros o incluso por ti mismo en el futuro.
- **Dificultad en Proyectos Colaborativos:** La coordinación manual de los cambios en los scripts y modelos se convierte en un cuello de botella constante, limitando el tamaño y la eficiencia de los equipos de ciencia de datos.

3. Introducción a Git: Historia y Conceptos Clave

3.1. ¿Qué es Git?

Git es un sistema distribuido de control de versiones (DVCS), gratuito y de código abierto, lanzado bajo licencia GPLv2. Fue diseñado y desarrollado originalmente por **Linus Torvalds** en 2005 para gestionar el desarrollo del núcleo de Linux, un proyecto masivo y altamente colaborativo.

3.2. Ventajas Fundamentales de Git

Linus Torvalds lo creó con objetivos muy específicos en mente, que hoy son sus mayores fortalezas:

- **Rendimiento Excepcional:** Git es increíblemente rápido. Las operaciones clave como ramificación, fusión y acceso al historial son casi instantáneas, incluso con repositorios muy grandes o con muchos archivos de datos.
- **Diseño Distribuido:** A diferencia de sistemas centralizados como Subversion (SVN), Git permite que cada científico de datos tenga una copia **completa** del repositorio, incluyendo todo el historial de scripts, notebooks y configuraciones. Esto significa que puedes trabajar sin conexión y tienes una copia de seguridad local de todo el proyecto.
- **Integridad de Datos:** Git está diseñado para garantizar la integridad de los datos de tu código. Utiliza sumas de comprobación (hashes SHA-1) para identificar y verificar cada cambio.

- **Ramificación y Fusión Robusta:** Las operaciones de ramificación y fusión son pilares en Git, diseñadas para ser fluidas y eficientes, incluso cuando diferentes personas trabajan en partes relacionadas de un mismo proyecto de datos.

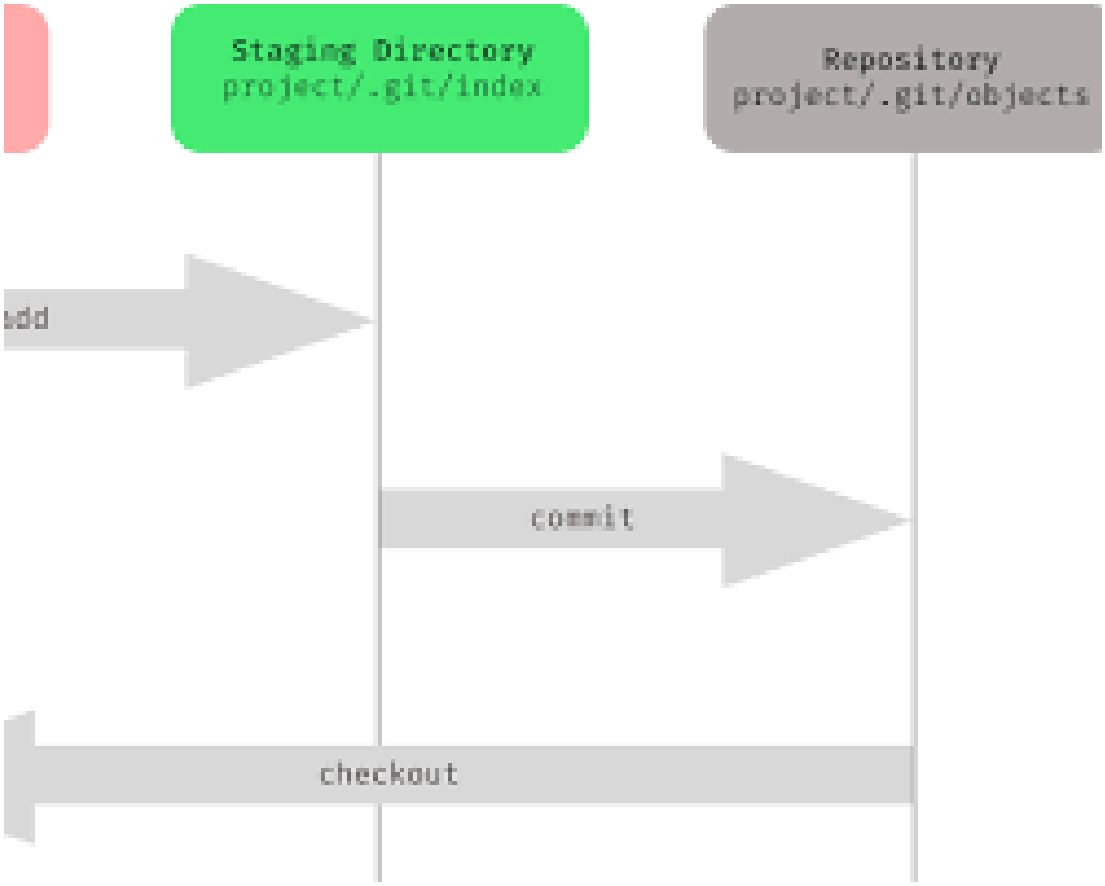
3.3. Git vs. GitHub/GitLab/Bitbucket

Es crucial entender que **Git no es lo mismo que GitHub, GitLab o Bitbucket**.

- **Git:** Es la **tecnología** subyacente, el sistema de control de versiones de línea de comandos que instalas en tu máquina. Permite la gestión de versiones de forma local.
- **GitHub/GitLab/Bitbucket:** Son **plataformas o servicios de alojamiento en la nube** que utilizan Git para gestionar repositorios remotos. Ofrecen una interfaz web amigable y añaden funcionalidades colaborativas más allá del control de versiones puro de Git, como seguimiento de incidencias, gestión de proyectos, Pull Requests (o Merge Requests en GitLab), Integración Continua/Despliegue Continuo (CI/CD), etc.

4. Los Tres Estados en Git

Git maneja tus archivos a través de un "ciclo de vida" que involucra tres estados principales. Comprender estos estados es fundamental para usar Git de manera efectiva.



1. **Directorio de Trabajo (Working Directory):**
    - Este es el área donde resides físicamente en tu sistema de archivos. Contiene tus scripts de Python (`.py`), notebooks Jupyter (`.ipynb`), archivos CSV o Parquet, o cualquier otro recurso de tu proyecto de ciencia de datos que hayas creado, modificado o eliminado.
    - Los archivos aquí pueden estar en dos sub-estados desde la perspectiva de Git:
        - **`Untracked (Sin rastrear)`**: Archivos nuevos (ej. un nuevo script de visualización, un dataset recién descargado) que Git aún no ha visto ni ha comenzado a seguir.
        - **`Modified (Modificado)`**: Archivos que Git ya está rastreando (ej. un script de preprocesamiento que has actualizado), pero que han sido cambiados desde el último `commit`.
2. **Área de Staging (Staging Area / Index / Área temporal transitoria):**
    - Es un área intermedia entre el directorio de trabajo y el repositorio local. Piensa en ella como una "bandeja de entrada" donde preparas los cambios específicos (ej. la versión final de un script de entrenamiento,

una actualización en un notebook) que quieres incluir en tu próximo `commit`.
    - Cuando añades archivos a esta área (`git add`), pasan al estado **`Staged (Preparado)`**. Esto significa que has seleccionado esos cambios específicos para ser guardados en la próxima "fotografía" (commit).
3. **Repositorio Local (Local Repository):**
    - Aquí es donde Git almacena todas las "fotografías" (commits) de tu proyecto de forma permanente en tu máquina local. Cada commit es una versión completa de tu código y recursos en ese momento.
    - Cuando realizas un `commit`, los cambios del área de staging se guardan en el repositorio local y el archivo pasa al estado **`Committed (Confirmado)`**.

El Ciclo de Vida de un Archivo de Datos en Git:

1. Creas o modificas un script de Python, un notebook Jupyter o un archivo de configuración en tu **Directorio de Trabajo** (estado: `Untracked` o `Modified`).
2. Le dices a Git qué cambios específicos quieres incluir en la siguiente "fotografía" de tu proyecto usando `git add`. Estos cambios se mueven al **Área de Staging** (estado: `Staged`).
3. Tomas la "fotografía" final de los cambios preparados usando `git commit`. Esta "fotografía" se guarda en tu **Repositorio Local** (estado: `Committed`).

5. Actividad Práctica (Laboratorio 1)

¡Manos a la obra! Es hora de configurar Git y dar los primeros pasos en la terminal, aplicándolos a un contexto de ciencia de datos.

5.1. Instalación de Git

Lo primero es asegurarnos de que Git está instalado en tu sistema.

Paso 1: Verificar si Git está instalado Abre tu terminal (o línea de comandos) y escribe:

```
Unset
git --version
```

- **Si ves un número de versión** (ej. `git version 2.30.1`), ¡felicidades! Git ya está instalado. Puedes saltar al punto 5.2.
- **Si ves un mensaje como `command not found: git`** o similar, significa que Git no está instalado o no está configurado en tu `PATH`. Sigue las instrucciones para tu sistema operativo:

Paso 2: Instalar Git según tu sistema operativo

- **macOS:**
    1. La forma más sencilla es instalar las Herramientas de Desarrollo de Xcode. Abre la terminal y ejecuta:

```
Unset

xcode-select --install
```

    2. Sigue las indicaciones en pantalla.
    3. Alternativamente, si usas Homebrew (un gestor de paquetes para macOS), puedes instalarlo con:

```
Unset

brew install git
```

- **Linux (Ubuntu/Debian):** Abre la terminal y ejecuta:

```
Unset

sudo apt-get update

sudo apt-get install git
```

- Para otras distribuciones (Fedora, CentOS, etc.), consulta la documentación oficial de Git para Linux: https://git-scm.com/download/linux
- **Windows:**

1. La forma más sencilla es descargar el instalador oficial de Git para Windows desde la página de Git SCM: https://git-scm.com/download/win
2. Ejecuta el archivo `.exe` descargado y sigue el asistente de instalación. Se recomienda aceptar la mayoría de las opciones por defecto, pero asegúrate de que se instale Git Bash, que es la terminal de estilo Unix que usaremos en el curso.

Paso 3: Confirmar la instalación Una vez finalizada la instalación, vuelve a ejecutar `git --version` para asegurarte de que todo está correcto.

5.2. Configuración Inicial de Git

Antes de empezar a usar Git en tus proyectos, debes decirle quién eres. Esta información se adjuntará a cada `commit` que realices.

Paso 1: Configurar tu nombre de usuario En la terminal, escribe:

```
Unset
git config --global user.name "Tu Nombre Científico"
```

Reemplaza `"Tu Nombre Científico"` con tu nombre real (ej. `"Dra. Ana García"`). Las comillas son necesarias si tu nombre tiene espacios.

Paso 2: Configurar tu dirección de correo electrónico Ahora, configura tu correo electrónico. Es recomendable usar el mismo correo que utilizarás en plataformas como GitHub o GitLab para que tus contribuciones se vinculen correctamente.

```
Unset
git config --global user.email
"tu.email.cientifico@example.com"
```

Reemplaza `"tu.email.cientifico@example.com"` con tu dirección de correo real.

Paso 3: Verificar tu configuración Para ver la configuración que has establecido (y otras configuraciones predeterminadas), puedes usar:

```
Unset
git config --list
```

Verás una lista de pares clave-valor, donde deberían aparecer `user.name` y `user.email`.

Paso 4: Configurar un editor de texto por defecto (Opcional, pero recomendado) Git a veces necesita abrir un editor de texto (por ejemplo, para mensajes de commit largos o resolución de conflictos). Por defecto, suele usar `Vim`, que puede ser complejo si no estás familiarizado. Puedes configurarlo para usar tu editor preferido, como Visual Studio Code o un editor de texto plano que uses para scripts.

Si usas Visual Studio Code (asegúrate de que el comando `code` está disponible en tu terminal):

```
Unset
git config --global core.editor "code --wait"
```

El `--wait` es importante para que Git espere a que cierres el editor antes de continuar.

Si usas Nano (un editor de terminal más sencillo):

```
Unset
git config --global core.editor "nano"
```

5.3. Primeros Pasos en la Terminal y con Git

Ahora que Git está instalado y configurado, vamos a crear nuestro primer repositorio local para un proyecto de análisis de datos.

Paso 1: Crear un directorio para tu primer proyecto de datos Abre tu terminal y crea una nueva carpeta para tu proyecto. Usaremos el comando `mkdir` (make directory).

```
Unset
mkdir proyecto-analisis-datos
```

Paso 2: Navegar a la nueva carpeta Usa el comando `cd` (change directory) para entrar en tu nueva carpeta:

```
Unset
cd proyecto-analisis-datos
```

Puedes verificar que estás en la carpeta correcta con `pwd` (print working directory) en Linux/macOS o `cd` sin argumentos en Windows.

Paso 3: Inicializar un repositorio Git local Dentro de la carpeta `proyecto-analisis-datos`, ejecuta el comando mágico que convierte cualquier directorio en un repositorio Git:

```
Unset
git init
```

Verás un mensaje como: `Initialized empty Git repository in /ruta/a/tu/proyecto-analisis-datos/.git/`.

Paso 4: Explorar el directorio `.git` Git ha creado una subcarpeta oculta llamada `.git` dentro de tu proyecto. Esta carpeta contiene toda la magia de Git (el historial, configuraciones, etc.). ¡Nunca la borres ni modifiques sus archivos directamente!

Para verla (en sistemas Unix como macOS/Linux, o Git Bash en Windows):

```
Unset
ls -al
```

Deberías ver `.git` en la lista.

Paso 5: Verificar el estado del repositorio El comando `git status` es tu mejor amigo para saber qué está pasando en tu repositorio.

```
Unset
git status
```

Deberías ver algo como:

```
Unset
On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to
track)
```

Esto te dice que estás en la rama `master` (o `main` si tu versión de Git es más reciente o la configuraste así), que aún no hay commits, y que no hay archivos para rastrear.

Paso 6: Crear tus primeros scripts de Python Usemos `touch` para crear archivos vacíos que representen tus scripts de trabajo (o puedes usar tu editor para crear archivos con contenido).

Ejemplo 1: Script de limpieza de datos

```
Unset
touch data_cleaning.py
```

Ejemplo 2: Script para entrenar un modelo

```
Unset
touch train_model.py
```

Paso 7: Verificar el estado nuevamente

```
Unset
git status
```

Ahora deberías ver algo como:

```
Unset
On branch master

No commits yet

Untracked files:

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


    data_cleaning.py

    train_model.py


nothing added to commit but untracked files present (use
"git add" to track)
```

Git te dice que `data_cleaning.py` y `train_model.py` son archivos **`Untracked`** (sin rastrear), lo que significa que Git los ve, pero no los está gestionando aún. Para que Git empiece a rastrearlos y considerar sus cambios, necesitamos el comando `git add`, que veremos en la próxima clase.

6. Posibles Errores y Soluciones

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

1. **Error: `command not found: git`**
    - **Causa:** Git no está instalado en tu sistema, o la ruta a su ejecutable no está incluida en la variable de entorno `PATH` de tu sistema.

    - **Solución:** Revisa las instrucciones de instalación para tu sistema operativo y asegúrate de haberlas seguido correctamente. Reinicia tu terminal después de la instalación.
2. **Error:** `fatal: not a git repository (or any of the parent directories): .git`
    - **Causa:** Estás intentando ejecutar un comando de Git (como `git status`) en un directorio que no es un repositorio de Git, o no estás en la carpeta raíz de un repositorio Git existente.
    - **Solución:**
        - Si es un proyecto nuevo, navega a tu carpeta de proyecto y ejecuta `git init` para inicializar el repositorio.
        - Si ya tienes un repositorio, asegúrate de que estás en la carpeta raíz del proyecto donde se encuentra la carpeta `.git` oculta. Usa `cd` para navegar.
3. **Error (macOS):** `xcrun: error: invalid active developer path (...)`
    - **Causa:** Después de una actualización de macOS, o por otras razones, las herramientas de línea de comandos de Xcode pueden haberse desvinculado o necesitar ser reinstaladas.
    - **Solución:** Reinstala las herramientas de línea de comandos de Xcode ejecutando en tu terminal:

```
Unset

xcode-select --install
```

    - Sigue las indicaciones en pantalla. Si persiste, reiniciar el ordenador o intentar `sudo xcode-select --reset` puede ayudar.

Conclusión de la Clase 1

¡Felicidades! Has completado la primera clase de nuestro curso de Git. Hoy hemos sentado las bases al entender qué es el control de versiones y por qué Git es una herramienta indispensable en el flujo de trabajo de un científico de datos. Hemos explorado los tres estados fundamentales de un archivo en Git y, lo más importante, has realizado la instalación y configuración inicial de Git en tu máquina, además de interactuar con la terminal para inicializar tu primer repositorio de análisis de datos.

En la próxima clase, profundizaremos en los comandos básicos para mover archivos entre estos estados y empezaremos a crear nuestras primeras "fotografías" de código: los `commits`.