Clase 10: Proyecto Final Colaborativo

Objetivos de la Clase

Al finalizar esta clase, integrarás y aplicarás todos los conocimientos adquiridos, siendo capaz de:

- Aplicar el **flujo de trabajo completo de Git** en un proyecto colaborativo de ciencia de datos.
- Trabajar en equipo utilizando **ramas, commits, Pull Requests y resolución de conflictos**.
- Practicar la **revisión de código** (scripts, notebooks) entre pares.
- Utilizar comandos avanzados y buenas prácticas en un **escenario práctico y simulado**.
- Consolidar la comprensión de Git, GitHub/GitLab en un **proyecto final**.

1. Introducción al Proyecto Final

Hemos recorrido un camino completo, desde los conceptos básicos de control de versiones hasta estrategias avanzadas y herramientas de línea de comandos. Ahora es el momento de poner a prueba todo lo aprendido en un **proyecto final colaborativo**.

Este proyecto simulará un escenario real de trabajo en equipo, donde cada uno contribuirá a un objetivo común utilizando las mejores prácticas de Git que hemos explorado.

Conceptos Clave a Aplicar:

- Inicialización y clonación de repositorios.
- Creación y gestión de ramas.
- Ciclo de vida de archivos (Untracked, Modified, Staged, Committed).
- Mensajes de commit semánticos y concisos.
- `git add`, `git commit`, `git status`, `git log`, `git diff`.
- Fusión de ramas (`merge`, `rebase`) y resolución de conflictos.
- Trabajo con repositorios remotos (`git fetch`, `git pull`, `git push`).
- Flujo de Pull Requests (PRs) y revisión de código.
- Uso de `.gitignore`.
- Comandos avanzados como `git stash`, `git revert`, `git cherry-pick` (si aplica).
- Productividad con alias (si los configuraste).

2. Propuesta de Proyecto: Análisis de Datos de Clientes

Escenario: Eres parte de un equipo de científicos de datos en una empresa de marketing. La tarea es realizar un **análisis exploratorio de datos (EDA) y una segmentación de clientes** para identificar grupos de usuarios con comportamientos similares. El proyecto se dividirá en varias fases y cada miembro del equipo será responsable de una parte.

Estructura del Proyecto (simulada):

- Un script principal (`main_analysis.py`).
- Un módulo para la carga y limpieza de datos (`data_loader.py`).
- Un módulo para el preprocesamiento de características (`feature_engineering.py`).
- Un módulo para la segmentación/clustering (`customer_segmentation.py`).
- Un notebook Jupyter para visualizaciones (`eda_notebook.ipynb`).
- Un archivo de configuración (`config.py`).
- Un `requirements.txt` para las dependencias.

Tareas Asignadas (simuladas para un equipo de 3-4):

- **Rol 1 (Líder/Integrador):**
  - Crear el repositorio inicial en GitHub/GitLab.
  - Establecer la estructura inicial de directorios y archivos base.
  - Revisar las Pull Requests de los demás.
  - Gestionar fusiones a `main`.
- **Rol 2 (Ingesta y Limpieza de Datos):**
  - Implementar la función de carga de datos en `data_loader.py`.
  - Añadir funciones de limpieza de datos (ej. manejo de nulos, eliminación de duplicados).
  - Actualizar `requirements.txt` con librerías necesarias.
- **Rol 3 (Ingeniería de Características):**
  - Implementar funciones de ingeniería de características en `feature_engineering.py` (ej. normalización, creación de nuevas variables).
  - Asegurarse de que el script principal pueda usar estas funciones.
- **Rol 4 (Modelado/Análisis Exploratorio):**
  - Implementar un algoritmo de clustering (ej. K-Means) en `customer_segmentation.py`.
  - Crear visualizaciones clave del EDA en `eda_notebook.ipynb`.

3. Flujo de Trabajo Colaborativo del Proyecto Final

Seguiremos un flujo de trabajo similar a **GitHub Flow**, centrado en Pull Requests, dado que es robusto y ampliamente utilizado en equipos.

3.1. Fase Inicial: Creación y Clonación del Repositorio

1. **Un miembro del equipo (Rol 1, Líder) crea el Repositorio Remoto:**
    - Ve a GitHub/GitLab y crea un **nuevo repositorio público** (ej. `analisis-clientes-ds`).
    - **¡Inicialízalo con un `README.md` y un `.gitignore` para Python!** Esto nos da una base inicial limpia. (En GitHub, puedes seleccionar "Add .gitignore" y elegir la plantilla "Python").
    - Copia la URL (SSH recomendada) del repositorio.
2. **Todos los miembros del equipo clonan el Repositorio Remoto:**
    - Abre tu terminal y navega a tu directorio de proyectos.

```
Unset
  git clone <URL_del_repositorio_remoto>
  cd analisis-clientes-ds
```

    - Verifica que el repositorio está clonado y que tu rama `main` local está actualizada: `git status`.

3.2. Desarrollo de Tareas Individuales (Ciclo de Feature Branch)

Cada miembro del equipo trabajará en su tarea asignada, siguiendo este ciclo:

1. **Asegúrate de que tu rama `main` local esté actualizada:**

```
Unset

git switch main
git pull origin main
```

2. **Crea una nueva rama para tu tarea/funcionalidad:**
    - Usa una convención de nombres descriptiva (ej. `feat/cargar-datos`, `feat/limpiar-duplicados`, `feat/normalizar-variables`, `feat/implementar-kmeans`, `feat/visualizar-eda`).

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

3. **Desarrolla tu código y haz commits pequeños y frecuentes:**
    - Modifica los archivos asignados (ej. `data_loader.py`, `feature_engineering.py`, etc.).
    - Añade contenido Python relevante y guárdalo.
    - Asegúrate de que tu `.gitignore` es adecuado para los archivos que generas (ej., resultados intermedios, entornos virtuales).
    - Haz commits regulares, utilizando **mensajes semánticos y descriptivos**:

```
Unset

git add . # O archivos específicos
git commit -m "feat(data): Implementar carga inicial de datos desde CSV"
# O: git commit -m "fix(eda): Corregir leyenda en gráfica de dispersión"
```

    - **Consejo:** Si tu tarea es grande, divídela en sub-tareas lógicas y haz un commit por cada una.
4. **Envía tu rama al repositorio remoto:**
    - Una vez que tengas un conjunto de commits que completan una parte lógica de tu tarea (o al final del día de trabajo).

```
Unset
  git push -u origin <nombre_de_tu_rama>
```

    - Si es la primera vez que envías esta rama, `-u` la configurará para seguimiento remoto.

3.3. Creación y Gestión de Pull Requests (PRs)

Una vez que tu trabajo en una rama está listo para ser revisado e integrado:

1. **Abre la Pull Request (desde la interfaz web de GitHub/GitLab):**
    - Ve a tu repositorio en la plataforma (GitHub/GitLab).
    - Verás un banner sugeriendo abrir una PR desde tu rama recién enviada. Haz clic en `Compare & pull request`.
    - **Establece la rama base (`main`) y la rama de comparación (tu rama).**
    - **Escribe un Título claro y una Descripción detallada de tu PR.**
        - Explica qué problema resuelve o qué funcionalidad añade.

        - Describe brevemente cómo lo implementaste.
        - Si aplica, menciona los resultados clave o visualizaciones.
        - Puedes incluir capturas de pantalla de gráficos o salidas si es un notebook.
    - Asigna revisores (ej., el Rol 1, Líder, u otros compañeros).
    - Haz clic en `Create pull request` (o `Create draft pull request` si aún no está completamente lista).

3.4. Revisión de Código y Colaboración (`Peer Review`)

Este es un paso crucial para asegurar la calidad y compartir conocimiento.

1. **Revisores (`Rol 1` o compañeros) acceden a la PR:**
    - Van a la pestaña `Pull requests` en el repositorio.
    - Revisan el título, descripción y, lo más importante, la pestaña `Files changed`.
2. **Proceso de Revisión:**
    - **Entender el Contexto:** Lee la descripción de la PR. ¿Qué se supone que hace?
    - **Análisis del Código:**
        - **Python:** ¿Es el código legible, bien comentado? ¿Sigue las convenciones de estilo? ¿Es eficiente?
        - **Data Science:** ¿La lógica de preprocesamiento es correcta? ¿El modelo se inicializa con los parámetros adecuados? ¿Se manejan los casos borde (ej., nulos, outliers)? ¿Se están guardando o cargando archivos de forma segura?
        - **Reproducibilidad:** ¿El script o notebook es reproducible? ¿Las dependencias están en `requirements.txt`?
    - **Comentarios y Sugerencias:**
        - Deja comentarios específicos en las líneas de código relevantes.
        - Sé constructivo y empático (Clase 6). Ofrece sugerencias en lugar de críticas directas.
        - Utiliza la función de "sugerencias" de GitHub para proponer cambios directos.
    - **Aprobar o Solicitar Cambios:** El revisor puede aprobar la PR o solicitar cambios si es necesario.

3.5. Resolución de Conflictos y Actualización de la PR

Si la rama `main` avanza mientras tu PR está abierta y hay cambios que colisionan con tu rama, la PR mostrará un conflicto.

1. **Identificar el Conflicto:** La plataforma te lo indicará ("This branch has conflicts that must be resolved").
2. **Actualizar tu Rama de PR y Resolver:**

- Cambia a tu rama local de la PR: `git switch <nombre_de_tu_rama>`
- Trae los cambios de `main` a tu rama de PR y resuelve los conflictos:

```
Unset

git pull origin main
# Resuelve los conflictos en tus archivos (igual que en la Clase 4)
git add .
git commit -m "fix(merge): Resolver conflictos con main"
```

- Envía los cambios actualizados a tu rama remota de PR:

```
Unset

git push origin <nombre_de_tu_rama>
```

- La Pull Request en GitHub se actualizará automáticamente y mostrará que los conflictos han sido resueltos.

3.6. Fusión de la Pull Request

Una vez que la PR ha sido revisada y aprobada (y no hay conflictos), se puede fusionar.

1. **Fusionar la PR (por el Rol 1, Líder, o quien tenga permisos):**
   - En la interfaz web de la PR, haz clic en `Merge pull request`.
   - Elige la opción de fusión adecuada (ej., `Merge commit` para mantener el historial completo, `Squash and merge` para un historial lineal en `main`).
   - Confirma la fusión.
2. **Eliminar la rama remota:** Después de la fusión, la plataforma te dará la opción de `Delete branch`. Haz clic en ella para mantener el repositorio remoto limpio.
3. **Actualizar tu `main` local y limpiar la rama local:**
   - Todos los miembros del equipo deben actualizar su rama `main` local:

```
Unset

git switch main
```

```
git pull origin main
```

    - Elimina la rama local de tu tarea (si ya se fusionó):

```
Unset

git branch -d <nombre_de_tu_rama>
```

4. Comandos y Conceptos Avanzados en el Proyecto Final (Opcional/Como Necesidad)

Durante el desarrollo del proyecto, podrías encontrarte con situaciones que requieran el uso de comandos avanzados.

- **`git stash`**: Si necesitas pausar tu trabajo actual en una rama para una tarea urgente en otra.
    - `git stash push -m "WIP: nombre de tarea"`
    - `git stash pop`
- **`git revert`**: Si un commit ya fusionado en `main` causa un problema y necesitas deshacerlo de forma segura sin reescribir el historial.
    - `git revert <hash_del_commit_problematico>`
- **`git cherry-pick`**: Si un compañero hizo una corrección crítica en su rama que aún no está lista para fusionarse, pero tú la necesitas ya en la tuya.
    - `git cherry-pick <hash_del_commit_de_correccion>`
- **Alias de Git**: ¡Utiliza los alias que configuraste en la Clase 9 (`git st`, `git ll`, `git co`, `git cm`) para acelerar tu flujo de trabajo!
- **`.gitignore`**: Asegúrate de que los archivos de entorno (`.env`), datasets intermedios, modelos entrenados (`.pkl`, `.h5`), y carpetas de resultados/logs están correctamente ignorados.

5. Posibles Errores y Soluciones durante el Proyecto Final

Trabajar en un proyecto colaborativo puede presentar desafíos. Aquí te presento algunos errores comunes y cómo abordarlos.

1. **Error: `Updates were rejected because the tip of your current branch is behind its remote counterpart.` (Push Rechazado).**
    - **Causa:** Alguien más ha empujado cambios a la misma rama que tú estás intentando empujar, y tu versión local no tiene esos cambios.
    - **Solución:** Siempre haz `git pull origin <rama_actual>` antes de intentar `git push`. Resuelve cualquier conflicto si surge, luego `git add .`, `git commit` (si hubo conflicto), y finalmente `git push`.
2. **Error: Múltiples conflictos en una Pull Request.**
    - **Causa:** Tu rama de característica ha divergido mucho de la rama `main` (por ejemplo, has trabajado en ella por mucho tiempo sin actualizarla).
    - **Solución:**
        1. **Frecuencia:** Haz `git pull origin main` en tu rama de característica con regularidad (ej., cada día, o antes de empezar una sesión de trabajo grande) para integrar los cambios de `main` y resolver pequeños conflictos a medida que surgen.
        2. **Rebase para limpiar:** Si tu historial local es un caos, puedes `git rebase main` en tu rama de característica *antes de abrir la PR* (solo si tu rama no ha sido compartida aún). Esto crea un historial más limpio para la PR.
3. **Error: "PR demasiado grande" o "demasiados cambios para revisar".**
    - **Causa:** Has empaquetado demasiadas funcionalidades o refactorizaciones en una sola Pull Request.
    - **Solución:** Sigue la buena práctica de "hacer una sola cosa" por PR (Clase 6). Divide la tarea en PRs más pequeñas y manejables. Utiliza Feature Toggles para desplegar código incompleto de forma segura.
4. **Problema: El modelo entrenado o los datos generados son demasiado grandes para Git.**
    - **Causa:** Git no está diseñado para versionar archivos binarios grandes (como modelos entrenados `.pkl`, datasets `.csv` de GBs, etc.). Estos archivos hinchan el repositorio y lo hacen lento.
    - **Solución: No versionar datasets o modelos grandes directamente en Git.**
        1. Utiliza `.gitignore` para excluirlos.
        2. Para el control de versiones de datos grandes, considera **Git LFS (Large File Storage)**, DVC (Data Version Control) o herramientas similares, que gestionan enlaces a archivos externos en lugar de almacenarlos directamente en Git. (Esto es un tema avanzado fuera del alcance del curso, pero importante para científicos de datos).
5. **Error: Credenciales de bases de datos o claves API subidas accidentalmente.**
    - **Causa:** Olvidaste añadir `secrets.env` o `credentials.json` a tu `.gitignore` antes de hacer el commit inicial.
    - **Solución:**
        1. **¡NO PANICES!** No es el fin del mundo.

2. **Revoca la credencial/clave API de inmediato.** Genera una nueva.
3. Añade el archivo al `.gitignore`: `echo "secrets.env" >> .gitignore`.
4. Haz `git rm --cached secrets.env` para que Git deje de rastrearlo.
5. Haz un commit: `git commit -m "fix(security): Eliminar credenciales sensibles del historial"`
6. **Para eliminarlo del historial completo (avanzado y solo si es crítico):** Necesitarías herramientas como `git filter-branch` o `BFG Repo-Cleaner` para reescribir el historial y eliminar el archivo por completo. **¡Esto es muy destructivo y debe usarse con extremo cuidado y coordinación de equipo!**

Conclusión y Próximos Pasos

¡Felicidades! Has completado el curso "Introducción a Git y Control de Versiones". Has pasado de ser un principiante a tener una comprensión sólida de las herramientas fundamentales que impulsan el desarrollo de software y la colaboración en ciencia de datos.

Este proyecto final te ha dado la oportunidad de integrar todos los conceptos: desde la gestión básica de versiones, la ramificación y la fusión, hasta las complejidades de la colaboración en equipo y la resolución de conflictos.

Tus próximos pasos para seguir creciendo:

- **¡Sigue practicando!** La maestría en Git viene con la práctica constante.
- **Explora más a fondo los Hooks de Git:** Para automatizar tareas de CI/CD en tu máquina local.
- **Aprende sobre Git LFS o DVC:** Si trabajas con datasets o modelos muy grandes que necesitan versionarse de forma eficiente.
- **Profundiza en CI/CD (Integración Continua / Despliegue Continuo):** Cómo GitHub Actions, GitLab CI/CD, Jenkins, etc., automatizan pruebas y despliegues cada vez que envías código.
- **Contribuye a Proyectos de Código Abierto:** Utiliza tus nuevas habilidades en proyectos reales y aprende de otros desarrolladores (Clase 6).
- **Mantente Actualizado:** El mundo de Git y DevOps evoluciona. Sigue blogs, foros y la documentación oficial.

¡El control de versiones es una habilidad fundamental que te abrirá muchas puertas en tu carrera como científico de datos! ¡Adelante!