Aprendizaje de máquina II – Carrera de especialización en inteligencia artificial

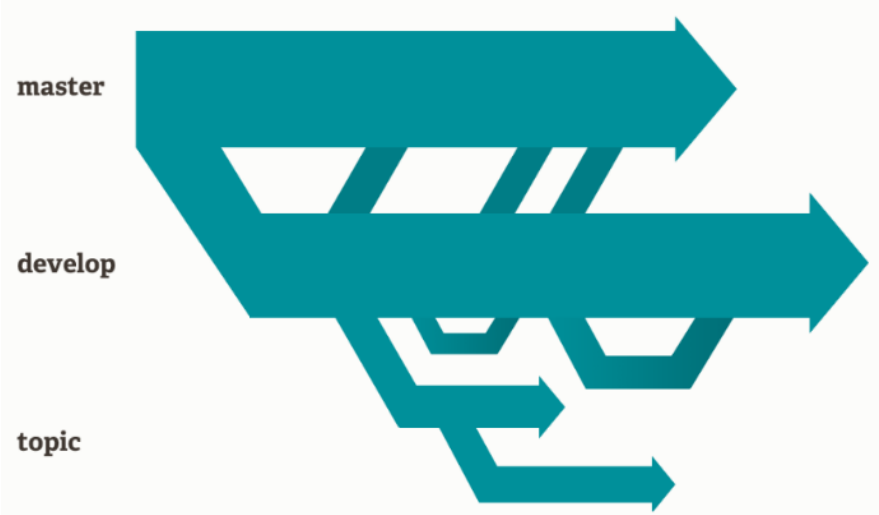
En este archivo se detallan algunos conceptos sobre el uso de Git (https://git-scm.com/) y GitHub (https://github.com/) para el trabajo de manera conjunta entre desarrolladores.

¿Qué es Git?

Git es una herramienta de código abierto para control de versiones de código fuente, diseñada para administrar desde pequeños hasta grandes proyectos con velocidad y eficiencia.

Una de las principales características que destaca a Git del resto de las herramientas para versionar código es la posibilidad de crear nuevas ramas (_branching_). Las ramas son réplicas del código existente que pueden ser modificadas sin alterar el código principal del proyecto.

Git permite (y recomienda) al usuario la creación de múltiples ramas locales que pueden ser completamente independientes una de otra. La creación, fusión (_merging_) y eliminación de todas estas ramas toma tan solo algunos segundos.



¿Por qué versionar el código?

El objetivo de versionar el código es tener un seguimiento de los cambios realizados en el proyecto.

Evitar archivos duplicados con nombres distintos y tener un historial de trabajo.

Pasaríamos de tener un directorio de trabajo así:

```
Notebooks/
├── modelo.ipynb
├── modelo_v2.ipynb
├── modelo_v3.ipynb
├── modelo_v3_final.ipynb
└── modelo_v3_final_final.ipynb
```

A uno así:

```
Notebooks/
└── modelo.ipynb
```

¿Cuál es la diferencia entre Git y GitHub?

Como mencionamos anteriormente Git es una herramienta que permite realizar un control de versiones de nuestro trabajo pero de manera local.

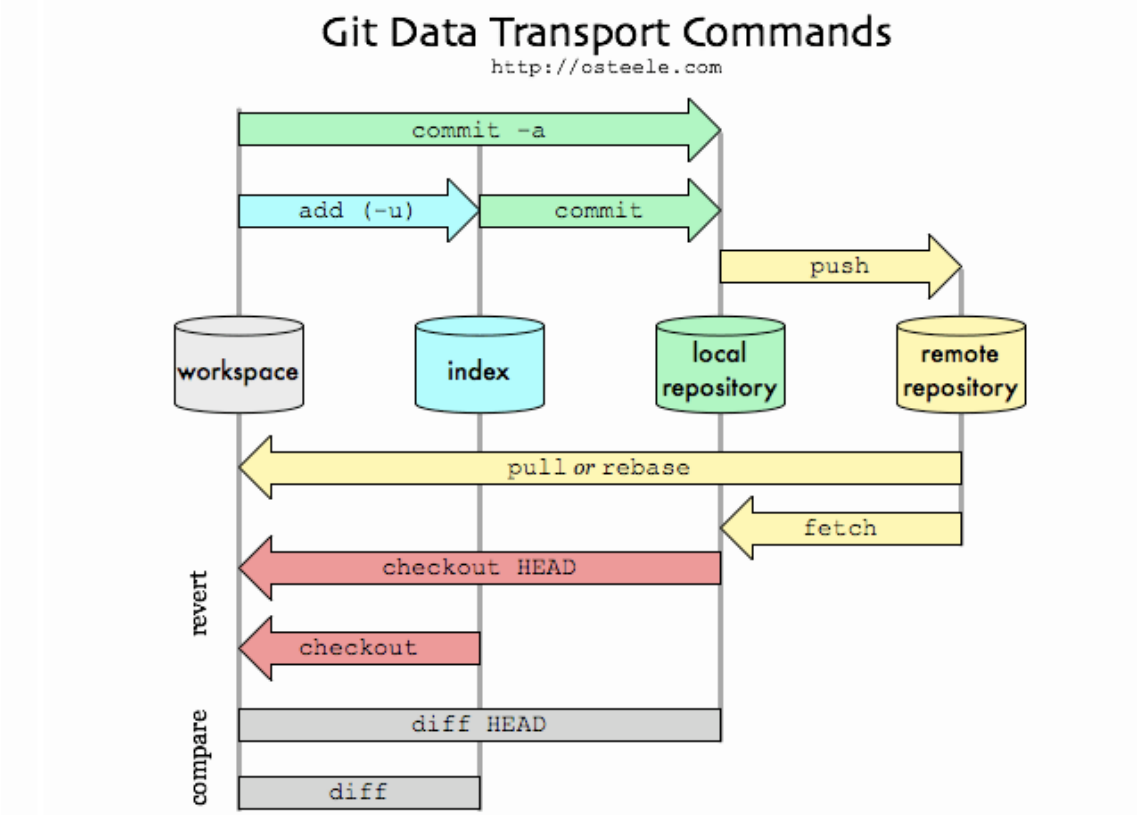
En caso de que nos encontremos trabajando con otras personas en un mismo proyecto, es conveniente que todos tengamos acceso a las versiones del código sin la necesidad de tener acceso a la PC donde se encuentra almacenado el código local.

Es ahí donde entra GitHub como un servidor remoto donde podremos subir las versiones de nuestro código.

Por otra parte, GitHub también ofrece una interfaz gráfica (GUI) muy amigable para realizar algunas de las tareas principales de Git.

Entonces, Git es una herramienta usada para administrar múltiples versiones de un código fuente que luego son transferidas a un repositorio de Git, mientras que GitHub sirve como un almacenamiento en la nube para subir copias de todo el repositorio de Git.

Estructura de trabajo en Git y principales comandos



Workspace -> Working directory

Index -> Staging area

Flujos de trabajo con Git

Existen distintas formas de administrar la creación de ramas dentro de un proyecto de software versionado en Git. Los distintos flujos de trabajo con ramas, buscan generar una metodología ordenada y óptima de trabajar de forma colaborativa en un proyecto.

Por ejemplo, podemos encontrar:

- GitFlow

- GitHub-Flow

- Trunk-Based development

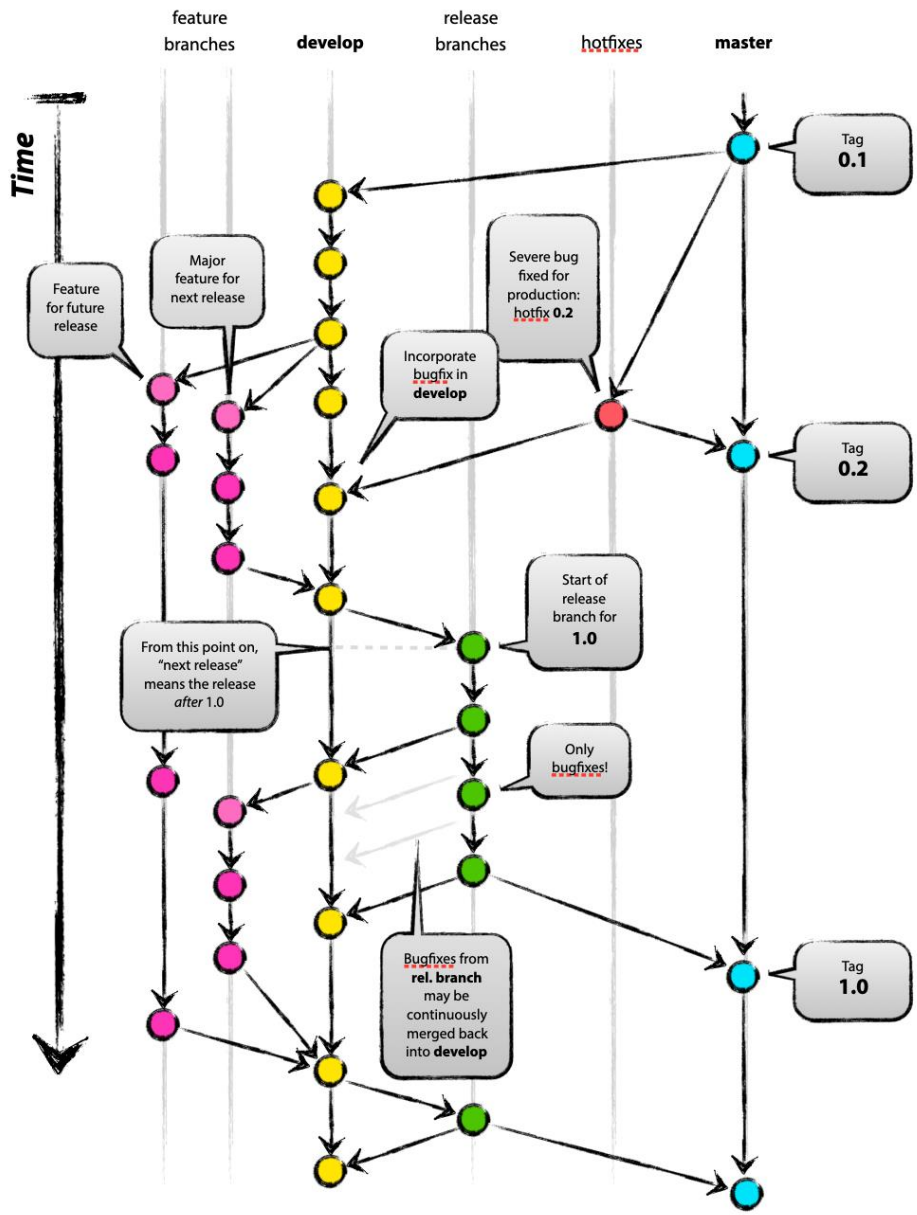
GitFlow [https://nvie.com/posts/a-successful-git-branching-model/]

Es una metodología desarrollada hace más de 10 años sobre cómo utilizar Git pero, en el último tiempo, fue reemplazada en algunos ámbitos de trabajo a medida que aparecieron otras metodologías más flexibles.

GitFlow consiste en trabajar en una rama develop que es una rama basada en la master/main. Al momento de generar una nueva feature, se crean ramas basadas en develop para desarrollar estas nuevas features (feat/new-feature-eng-pre-processing) que luego son unidas (merge) con la rama develop.

Normalmente no se trabaja sobre la rama master/main, se trabaja sobre la rama de desarrollo y una vez que el código está listo para llevarse a producción, pasa a una rama llamada release en donde se revisan los próximos merge con la rama main y se corrigen bugs encontrados.

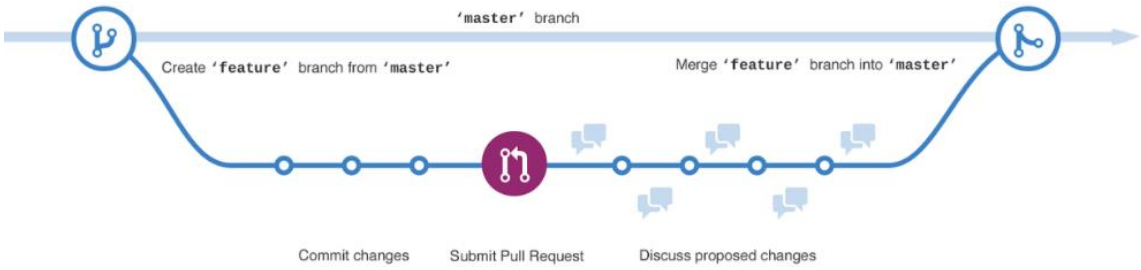
También existe una rama denominada hotfixes que se utiliza para cambios que deben salir rápidamente a producción.



GitHub Flow [https://docs.github.com/es/get-started/quickstart/github-flow]

GitHub Flow es una metodología de trabajo más flexible que Git Flow y hace uso de algunas características que provee GitHub.

En esta metodología hay una única rama main en donde se van agregando las nuevas features desarrolladas. Para agregar una nueva feature, se genera una nueva rama basada en la rama principal, en donde se realizan las modificaciones (commits), luego se solicita un Pull Request para unir (merge) esta nueva rama a la principal y comienza un proceso de code review entre el desarrollador y el revisor del código. Cuando ambas partes están de acuerdo en el código de la nueva feature, se realiza la fusión con la rama principal.

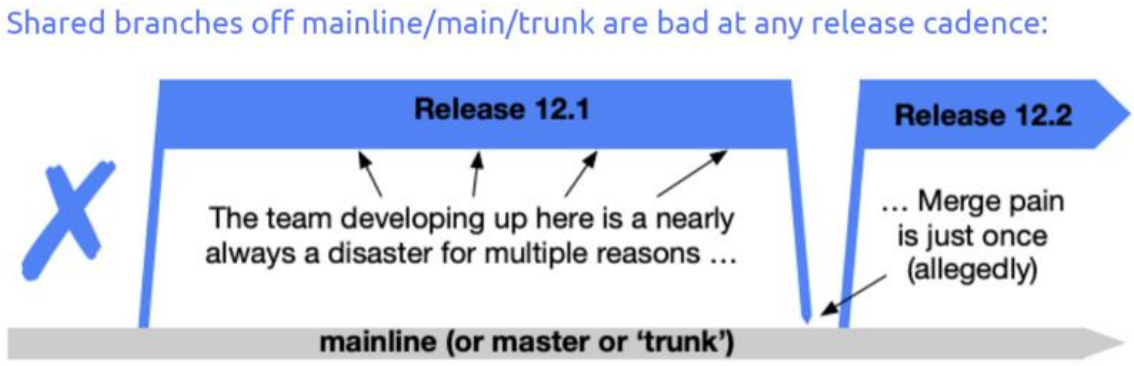


Idealmente, **cada nueva rama/modificación contiene un cambio completo y aislado**. Esto facilita volver atrás las modificaciones realizadas si se decide adoptar otro enfoque. Por ejemplo, si queremos renombrar una variable y agregar algunas pruebas, se debe poner el nuevo nombre de la variable en un commit y las pruebas en otro. Posteriormente, si queremos mantener las pruebas pero también deseamos revertir el renombramiento de variable, podemos revertir el commit específico que contenía dicho renombramiento. Si ponemos el renombre de variable y las pruebas en el mismo commit o si propagamos el renombre de variable a través de varios commits, sería necesario más esfuerzo para revertir los cambios.

Trunk-based development [https://trunkbaseddevelopment.com/]

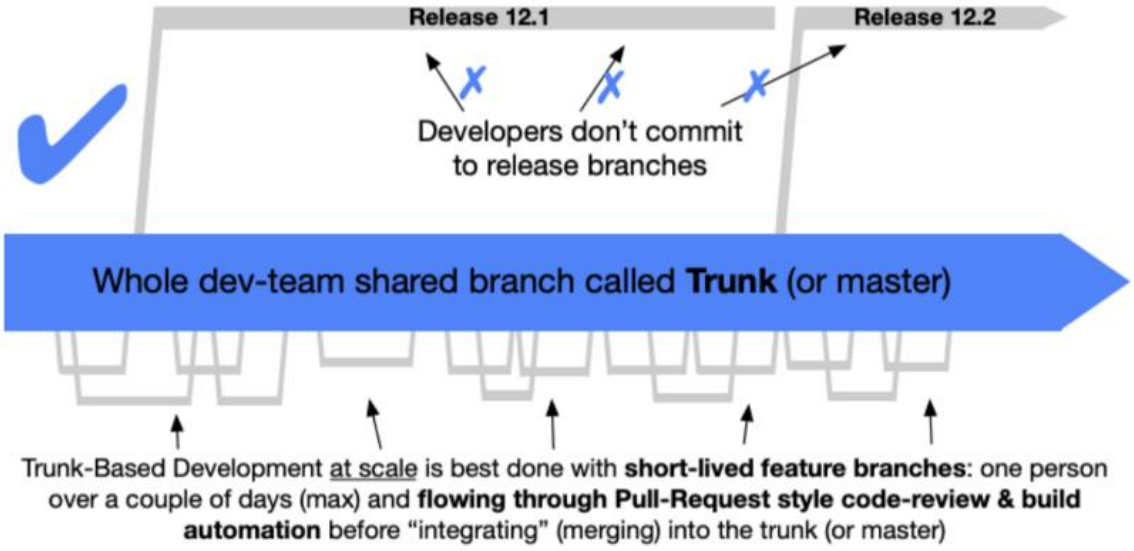
Al igual que GitHub Flow se trabaja con una misma rama. La diferencia es que establece mayores condiciones para trabajar sobre esta rama principal.

El objetivo es pasar de una forma de trabajo así:



A una forma de trabajo así:

Scaled Trunk-Based Development:



En esta metodología no se crean ramas de release como en el caso de GitFlow si no que los desarrolladores van agregando su código a la rama trunk/main.

La duración de las nuevas ramas no puede ser mayor a algunos días, por lo general dos o tres. Algunos equipos de trabajo buscan subir código al repositorio a diario para evitar perder la facilidad de integrarlo.

Esta forma de trabajo más incremental ayuda mucho para definir el alcance de cada uno de los trabajos, estimar tiempos y distintos recursos.

El código de la rama trunk siempre tiene que estar listo para estar en producción en cualquier momento. Esto produce que se preste más atención a la aparición de bugs en los desarrollos.

La filosofía Trunk-based development requiere un proceso de CI para poder probar todo el código con el objetivo de saber si funciona correctamente antes de fusionarlo.

Conventional commits [https://www.conventionalcommits.org/en/v1.0.0/]

Es una convención sencilla sobre cómo escribir mensajes de commit en repositorios de Git. Provee un conjunto de reglas sencillas para crear un histórico de commits.

La estructura del mensaje de commit es la siguiente:

```
<type>[optional scope]: <description>
[optional body]
[optional footer(s)]
```

El mensaje está compuesto por los siguientes elementos:

1. **Fix**: este tipo de commit repara un bug en el código fuente (Corresponde a un PATCH en el versionado semántico v x.x.1).

2. **Feat**: este tipo de commit introduce una nueva característica o funcionalidad en el código fuente (corresponde a un cambio MINOR en el versionado semántico v x.1.x).

3. **BREAKING CHANGE/!**: este tipo de commit introduce un cambio considerable en el código fuente que generaría incompatibilidad entre versiones previas del desarrollo y la actual (corresponde a un cambio MAJOR en el versionado semántico v 1.x.x).

4. Otros como build:, chore:, ci:, docs:, etc.

El pie del mensaje es opcional, en el caso de un BREAKING CHANGE podría ser: BREAKING CHANGE: <alguna descripción>.