

Guía de Configuración de Entorno de Data Science con Python y R en Windows (WSL2 + VS Code)

Requisitos Previos

Antes de comenzar, asegúrate de contar con los siguientes requisitos mínimos y recomendados:

- **Sistema Operativo:** Windows 10 (versión 2004 o superior) o Windows 11, con soporte para WSL2 ¹. Es imprescindible un sistema de 64 bits con la virtualización habilitada en la BIOS ².
- **Hardware:** Se recomienda un procesador multinúcleo moderno y al menos 8 GB de RAM (16 GB o más ideal para cargas de trabajo intensivas en datos y entrenamiento de modelos). También es aconsejable disponer de una unidad SSD con espacio libre amplio (decenas de GB) para instalar paquetes, datos y contenedores Docker.
- **Software:** Visual Studio Code (última versión) instalado en Windows. Contarás con la extensión *Remote - WSL* para VS Code (se detalla su instalación más adelante). Asimismo, necesitarás una distribución Linux (ej. Ubuntu) instalada en WSL2 y actualizada.
- **Versiones de Python y R:** Python 3 (se recomienda la versión más reciente, p. ej. 3.10 o superior) y R (versión 4.x) para asegurar compatibilidad con las herramientas modernas.
- **Conexión a Internet:** necesaria para descargar paquetes, contenedores Docker y otras dependencias durante la instalación y configuración inicial.

Con los requisitos anteriores cubiertos, podrás proceder a instalar WSL2 y a configurar el entorno de desarrollo.

Instalación de WSL 2, Ubuntu y Configuración Inicial

Para instalar el Subsistema de Windows para Linux 2 (WSL 2) con Ubuntu, sigue estos pasos:

1. **Habilitar WSL 2 en Windows:** Abre una ventana de PowerShell **como administrador** y ejecuta el comando de instalación rápida de WSL:

```
wsl --install
```

Este comando habilita las características necesarias (Plataforma de Máquina Virtual y WSL) e instala por defecto la distribución Ubuntu ³. Al finalizar, reinicia el equipo si así lo indica la consola. (Nota: En Windows 10 antiguos puede ser necesario habilitar manualmente las características mediante `dism.exe` como se explica en la documentación, pero en versiones recientes `wsl --install` realiza todos los pasos automáticamente.)

1. **Elegir distribución Linux (opcional):** Por defecto se instala Ubuntu. Si deseas otra distribución (Debian, Kali, etc.), puedes listar las disponibles con `wsl --list --online` y luego instalar una específica con `wsl --install -d <NombreDistro>` ⁴.

2. **Establecer WSL 2 como versión predeterminada:** En la mayoría de sistemas actuales, WSL 2 será el valor por defecto ⁵. Si fuera necesario (por ejemplo, en Windows 10 actualizado desde versiones antiguas), fuerza WSL 2 como versión predeterminada con:

```
wsl --set-default-version 2
```

Si al ejecutar este comando aparece un mensaje indicando que falta el kernel de WSL 2, descarga e instala la actualización del kernel desde el enlace oficial (aka.ms/wsl2kernel) antes de continuar ⁶.

1. **Iniciar Ubuntu y crear usuario:** Tras reiniciar, inicia la aplicación "Ubuntu" recién instalada (desde el menú Inicio o ejecutando `wsl` en PowerShell). En el primer inicio, se abrirá una consola de Ubuntu solicitando que definas un nombre de usuario *Linux* (distinto del de Windows, puede ser tu nombre) y una contraseña. Introduce credenciales para crear tu usuario normal en la distro Linux ⁷. **Importante:** recuerda esa contraseña, ya que la necesitarás para usar `sudo` (privilegios administrativos en Linux).

2. **Actualizar el sistema Ubuntu:** Una vez en la consola de Ubuntu (WSL), es recomendable actualizar los paquetes a la última versión. Ejecuta:

```
sudo apt update && sudo apt upgrade -y
```

Esto refresca los repositorios e instala actualizaciones disponibles.

1. **Instalar herramientas de desarrollo esenciales:** Para trabajar con Python y R, instala paquetes básicos de compilación y utilidades. Por ejemplo:


```
sudo apt install -y build-essential git curl wget libssl-dev libffi-dev
```

- `build-essential` proporciona compiladores (gcc/g++), necesarios para construir paquetes Python (vía pip/poetry) o R desde fuente.
- `git` permitirá el control de versiones.
- Otras librerías como `libssl-dev`, `libffi-dev` u otras (`zlib1g-dev`, `libxml2-dev`, etc.) pueden ser requeridas al compilar ciertas dependencias de Python o R. Tenerlas instaladas previene errores de compilación más adelante.

Con WSL2 y Ubuntu instalados y actualizados, ya dispones de un entorno Linux base dentro de Windows. En los siguientes pasos configuraremos Visual Studio Code y las herramientas específicas de Python y R en este entorno.

Configuración de Visual Studio Code con Remote - WSL

Con WSL2 operativo, puedes usar Visual Studio Code para desarrollar directamente en el entorno Linux. Los pasos para configurar VS Code con WSL son:

1. **Instalar VS Code en Windows:** Descarga e instala Visual Studio Code (versión Windows x64) desde la página oficial. Durante la instalación, activa la opción "Add to PATH" para poder lanzar VS Code desde la línea de comandos con `code` ⁸.
2. **Instalar la extensión Remote - WSL:** En VS Code, abre la pestaña de extensiones y busca **Remote Development** (un paquete de extensiones oficial de Microsoft). Instala el paquete *Remote Development*, el cual incluye la extensión *Remote - WSL* (además de Remote SSH y Remote Containers) ⁹. Esta extensión permite que VS Code se ejecute dentro de WSL2 ¹⁰, de modo que puedas abrir carpetas y archivos que residen en Linux directamente desde VS Code en Windows.
3. **Abrir una sesión VS Code en WSL:** Existen dos maneras comunes:
4. **Desde la terminal WSL:** Navega hasta la carpeta de tu proyecto (ejemplo: `cd ~/mi-proyecto`) y ejecuta `code .`. Este comando abrirá VS Code conectado a tu entorno Ubuntu en WSL, abriendo la carpeta actual ¹¹. La primera vez, VS Code instalará su servidor interno en WSL automáticamente.
5. **Desde VS Code (Windows):** Inicia VS Code en Windows sin abrir carpeta, presiona `Ctrl+Shift+P` y elige la opción **"WSL: Connect to WSL"** o **"WSL: Reopen Folder in WSL"**. También puedes seleccionar la distribución (si tuvieras más de una) con **"WSL: New WSL Window using Distro..."**. Esto abrirá una nueva ventana de VS Code ya conectada al entorno Ubuntu WSL, lista para abrir proyectos allí ¹².
6. **Administración de extensiones en WSL:** Ten en cuenta que al usar VS Code con WSL, existe una arquitectura cliente-servidor: la interfaz de VS Code corre en Windows, pero tus herramientas, extensiones de desarrollo, terminal y archivos residen en Linux ¹³. Por ello, ciertas extensiones deben instalarse dentro de WSL. Por ejemplo, la extensión de Python, R o Docker deben aparecer instaladas "en WSL" para funcionar correctamente en ese entorno ¹⁴. VS Code normalmente te avisará (con un ícono ) si una extensión que tienes en Windows requiere instalación en WSL; puedes hacer clic en "Install in WSL" en la vista de Extensiones. Asegúrate de instalar en WSL todas las extensiones relevantes:
7. **Python** (Microsoft) – para soporte de Python (linting, debugging, Jupyter notebooks, etc).
8. **R** (si trabajarás con R, extensión *R* de Ikuadeu) – para soporte de lenguaje R.
9. **Docker** – para integración con Docker desde VS Code.
10. **Quarto** (Posit) – útil si vas a trabajar con documentos .qmd o R Markdown (lo veremos en sección de Quarto).

Igualmente, puedes instalar temas, iconos u otras extensiones de interfaz solo en Windows (no es necesario duplicarlas en WSL, ya que no interactúan con el entorno remoto).

Con esta configuración, Visual Studio Code actúa como tu IDE principal, editando archivos dentro de WSL2 y aprovechando funcionalidades como IntelliSense, depuración y control de versiones de manera transparente entre Windows y Linux ¹⁵ ¹⁶.

Entorno Python: Instalación y Configuración de Herramientas Esenciales

En el entorno Ubuntu WSL configurado, instalaremos ahora las herramientas clave para trabajar con Python de forma eficiente en proyectos de ciencia de datos y ML. Estas incluyen gestores de versiones y entornos, instaladores de paquetes, formateadores de código, linters y más.

pyenv: Gestor de Versiones de Python

pyenv es una utilidad que permite instalar y alternar fácilmente entre múltiples versiones de Python en un mismo sistema ¹⁷. Con pyenv puedes tener, por ejemplo, Python 3.8, 3.10 y 3.11 coexistiendo, y elegir cuál usar globalmente o por proyecto. Esto es invaluable para probar compatibilidad o usar características específicas de ciertas versiones.

Instalación: Para instalar pyenv en Ubuntu WSL, puedes usar el instalador automático:

```
curl https://pyenv.run | bash
```

Este script descargará pyenv en `~/.pyenv` y configurará los scripts necesarios. Tras ejecutarlo, añade las siguientes líneas al final de tu fichero `~/.bashrc` (si el instalador no lo hizo automáticamente):

```
export PYENV_ROOT="$HOME/.pyenv"
export PATH="$PYENV_ROOT/bin:$PATH"
eval "$(pyenv init -)"
```

Luego recarga la configuración (`source ~/.bashrc`) o abre una nueva terminal. Para verificar, ejecuta `pyenv --version`, que debería mostrar la versión de pyenv instalada.

Uso básico: - Instala una versión de Python con `pyenv install`. Por ejemplo, para instalar la última versión de la serie 3.11: `pyenv install 3.11`. (Puedes listar versiones disponibles con `pyenv install -l`). - Establece una versión global por defecto: `pyenv global 3.11.4` (ejemplo de versión específica). Esto hace que `python` apunte a esa versión globalmente. - Alternativamente, usa `pyenv local <versión>` dentro de un directorio de proyecto para fijar la versión de Python solo en ese proyecto (crea un fichero `.python-version` en la carpeta). - Tras instalar nuevas versiones, puedes usar `pyenv versions` para ver las instaladas y `python --version` para verificar cuál está en uso.

Con pyenv, el Python del sistema queda aislado: siempre puedes volver a él con `pyenv global system` si lo necesitas. pyenv no gestiona entornos virtuales por sí mismo, pero se integra bien con herramientas como `virtualenv` o managers de proyectos como Poetry.

pipx: Instalación Aislada de Aplicaciones Python

pipx es una herramienta para instalar aplicaciones de línea de comando de Python de forma **aislada** del entorno principal ¹⁸. Es decir, para utilidades como `black`, `flake8`, `httpie`, `you-get`, etc., pipx crea entornos virtuales separados y los instala allí, haciendo disponibles los comandos en tu PATH de usuario ¹⁹. Esto evita conflictos de dependencias entre herramientas y mantiene limpio tu entorno base (no más `sudo pip install` global).

Instalación: En Ubuntu 22.04+ puedes instalar pipx con apt:

```
sudo apt install -y pipx
pipx ensurepath
```

Asegúrate de que `~/.local/bin` esté en tu PATH (pipx lo añade con `ensurepath` para el usuario actual). Alternativamente, si pipx no está en los repos, puedes instalarlo con `python3 -m pip install --user pipx` y luego `pipx ensurepath`.

Uso básico: Supongamos que quieres instalar la herramienta `httpie` (un cliente HTTP de consola):

```
pipx install httpie
```

Esto descargará el paquete desde PyPI, creará un virtualenv aislado en `~/.local/pipx/venvs/httpie` y expondrá el comando `http` en tu PATH. Puedes luego ejecutar `http https://api.github.com` por ejemplo, sin preocuparte de que interfiera con otras librerías Python.

Con pipx instalaremos varias de las herramientas a continuación para mantenerlas separadas del entorno principal.

Poetry y uv: Gestión de Entornos Virtuales y Dependencias de Proyecto

Para gestionar las dependencias de cada proyecto Python y aislar su entorno, existen herramientas de alto nivel como **Poetry** y **uv**. Ambas utilizan el estándar `pyproject.toml` para definir dependencias y permiten reproducir entornos fácilmente a partir de archivos de bloqueo (lockfiles).

- **Poetry:** Es un gestor de dependencias y empaquetado para Python, muy popular en los últimos años. Con Poetry puedes definir las librerías requeridas en un proyecto (y sus versiones) en un fichero `pyproject.toml` y Poetry se encarga de crear un entorno virtual aislado y resolver las versiones compatibles, generando un `poetry.lock` para fijar las versiones exactas instaladas ²⁰ ²¹. Poetry también facilita la publicación de paquetes propios. La forma recomendada de instalar Poetry es a través de pipx (evitando su script oficial), por ejemplo:

```
pipx install poetry
```

De este modo Poetry quedará instalado en un entorno aislado (así no se rompe si actualizas la versión global de Python) ²². Tras instalarlo, puedes inicializar un nuevo proyecto con `poetry init` interactivo, añadir dependencias con `poetry add <paquete>` (usar `-D` o `--group dev` para dependencias de desarrollo), instalar todas las dependencias con `poetry install` y ejecutar comandos dentro del entorno con `poetry run <comando>` o abrir un shell con `poetry shell`. Poetry creará automáticamente un entorno virtual (normalmente ubicado en `~/.cache/pypoetry/virtualenvs/`).

- **uv**: Es una herramienta más reciente, escrita en Rust y publicada por la empresa Astral (creadores del formateador `ruff`). uv se presenta como un "todo en uno" rápido para gestionar proyectos Python: su objetivo es reemplazar a pip, pipx, virtualenv, pip-tools y Poetry con una sola interfaz ²³. De hecho, uv puede instalar Python (similar a pyenv), administrar entornos, instalar aplicaciones CLI (reemplazando pipx) y gestionar dependencias de proyectos con un lockfile único para todos los entornos. Es notablemente rápido (hasta 10-100 veces más que pip en instalación de paquetes, gracias a su implementación en Rust) ²³. Para instalar uv, también podemos usar pipx:

```
pipx install uv
```

(uv está disponible en PyPI, facilitando su instalación ²⁴). Una vez instalado, el flujo de trabajo con uv es similar:

- Iniciar un nuevo proyecto: `uv init` (crea `pyproject.toml` interactivo, similar a `poetry init`).
- Agregar dependencias: `uv add <paquete>` (usa `--dev` para dependencias de desarrollo).
- Instalar (sync) dependencias: `uv install` o simplemente usar `uv run` que automáticamente asegura que el entorno esté sincronizado.
- Ejecutar comandos: `uv run <comando>` (ejecuta dentro del entorno del proyecto).

uv administra automáticamente un entorno virtual por proyecto en el directorio `.uv` (dentro del proyecto, por defecto). Ofrece ventajas de velocidad y un enfoque unificado. Sin embargo, al ser relativamente nuevo, tiene algunas diferencias: por ejemplo, no soporta múltiples *groups* de dependencias separadas para dev/prod/test como sí lo hace Poetry ²⁵ — en uv todas las dependencias se instalan juntas (las marcadas como dev solo se excluyen al publicar un paquete). Aún así, uv está ganando adopción rápidamente por su eficiencia.

¿Poetry o uv? Ambas herramientas cumplen propósitos similares. Poetry es madura y ampliamente usada; uv es muy veloz y promete simplificar la gestión sustituyendo varias utilidades a la vez. Puedes elegir una u otra según tus preferencias. Por ejemplo, si ya tienes experiencia con Poetry, sigue con Poetry; si valoras el rendimiento y un workflow más sencillo, puedes probar uv ²⁶. En cualquier caso, usar uno de estos gestores es altamente recomendable para mantener proyectos aislados y con dependencias reproducibles.

Herramientas de Formateo, Linter y Calidad de Código

Para mantener un código Python limpio, consistente y libre de errores comunes, se utilizan las siguientes herramientas en el flujo de trabajo:

- **Black:** formateador automático de código. Configurándolo, Black reescribe tu código a un estilo estándar (PEP 8) cada vez que lo ejecutas, evitando discusiones de estilo en el equipo. Su filosofía es tener "una única manera de formatear" — al aplicarlo regularmente, el dif de cambios se reduce solo a cuestiones lógicas, nunca de formato.
- **isort:** organizador automático de imports. isort ordena las importaciones alfabéticamente y separa los grupos (built-in, terceros, locales) de forma consistente.
- **Flake8:** es una herramienta de linting que combina PyFlakes, pycodestyle y mccabe, detectando errores de sintaxis, malas prácticas y violaciones de estilo PEP8 que un formateador no corrige (por ejemplo, variables definidas pero no usadas, etc).
- **mypy:** es un analizador de tipos estático para Python. Si anotas tu código con *type hints*, mypy comprueba coherencia de tipos sin ejecutar el programa, capturando una clase de errores antes de tiempo (ej: pasar un `str` donde se espera un `int`, etc).
- **pre-commit:** es una herramienta para gestionar *hooks* de Git pre-compilación. Permite definir una serie de acciones que se ejecuten automáticamente **antes** de cada commit, por ejemplo, ejecutar Black, isort y flake8 para asegurar que el código que vas a commitear está formateado y libre de errores simples. De este modo, mantienes la calidad de código de forma consistente en cada commit.

Instalación: Todas estas herramientas se pueden instalar de forma global con pipx, o bien agregarlas como dependencias de desarrollo en Poetry/uv para cada proyecto. Una aproximación común es instalarlas con pipx para poder usarlas en cualquier directorio/proyecto. Por ejemplo:

```
pipx install black
pipx install isort
pipx install flake8
pipx install mypy
pipx install pre-commit
```

Verás que cada comando quedará disponible globalmente (`black --version`, `flake8 --version`, etc., deben responder una vez instalados).

Configuración y uso: - Puedes ejecutar Black e isort manualmente sobre tus archivos (`black .` formateará todo el proyecto, e `isort .` ordenará imports). - Flake8 y mypy se pueden correr para analizar el código (`flake8 .` / `mypy .`) y ver reportes de problemas. - pre-commit se configura añadiendo un archivo `.pre-commit-config.yaml` en el repositorio, listando los hooks (por ejemplo, usar los hooks oficiales de black, isort, flake8, mypy, incluso hooks para verificar que los notebooks están limpios de outputs, etc.). Luego ejecutas `pre-commit install` una vez en el repo para habilitar los hooks de git ²⁷. A partir de entonces, cada `git commit` disparará automáticamente las herramientas: típicamente, formateará el código y evitará el commit si hay errores de lint o tipos, hasta que se corrijan. - Todas estas herramientas son altamente configurables mediante ficheros de configuración: `pyproject.toml` o específicos como `black.toml`, `flake8.ini`, `mypy.ini` etc. Por ejemplo,

puedes configurar el line-length que usa Black, reglas a ignorar en flake8, estilos de import en isort, directorios a excluir, etc. Adaptar estas configuraciones a las guías de tu proyecto es recomendable.

Integrar estas utilidades en tu flujo de trabajo mejora enormemente la calidad y consistencia del código. Un flujo típico sería: al guardar, Black e isort pueden formatear automáticamente (VS Code puede configurarse para ejecutar formateador al guardar); antes de hacer commit, pre-commit ejecuta nuevamente Black/isort (asegurando que nada quedó sin formatear), flake8 y mypy revisan el código, y si algo falla, el commit se aborta para que lo corrijas. Así, cualquier error evidente se corrige antes de entrar al repositorio.

Entorno R: Instalación y Configuración de Herramientas Esenciales

Ahora configuraremos R dentro de WSL para proyectos de análisis de datos y estadística, integrándolo con VS Code.

Instalación de R en Ubuntu (WSL)

Para instalar R en Ubuntu (WSL), la forma más sencilla es usar los repositorios apt:

```
sudo apt update
sudo apt install -y r-base r-base-dev
```

Esto instalará la última versión de R disponible en los repositorios de tu versión de Ubuntu, junto con las herramientas de desarrollo (`r-base-dev`) necesarias para compilar paquetes R desde código fuente ²⁸. *Nota:* Si deseas la versión más reciente de R y tu distribución no la incluye, puedes añadir el repositorio oficial de CRAN. Por ejemplo, para Ubuntu 22.04 *Jammy*, añadir una línea como `deb https://cloud.r-project.org/bin/linux/ubuntu jammy-cran40/` en `/etc/apt/sources.list` (y agregar la clave GPG de CRAN) te permitirá instalar R 4.2+ desde CRAN ²⁹. Tras agregar, ejecutar de nuevo `apt update` e instalar `r-base`. (Consulta la documentación de CRAN para instrucciones detalladas).

Una vez instalado, puedes lanzar R escribiendo `R` en la terminal de Ubuntu. Verás el interprete de R funcionando en WSL.

languageserver: Autocompletado y Soporte de Lenguaje R en VS Code

Para que VS Code brinde funcionalidades de IDE (autocompletado, help, salto a definiciones, etc.) para R, se utiliza el **R Language Server**. Este servidor LSP es proporcionado por el paquete `languageserver` de R. Debes instalarlo dentro de R (en WSL):

1. Abre una sesión de R (ejecuta `R` en la terminal).
2. En la consola de R, instala el paquete ejecutando:

```
install.packages("languageserver")
```


Este paquete implementa el Language Server Protocol para R ³⁰. La extensión de R en VS Code lo utilizará automáticamente para ofrecer autocompletado inteligente, información de funciones, diagnóstico de errores de sintaxis, etc.

A continuación, en VS Code, asegúrate de tener instalada la **extensión R** (de Ikuyadeu) ³¹. Si la instalaste previamente en Windows, verifica en la pestaña de Extensiones (como mencionamos en la sección de VS Code) que también esté instalada en el entorno WSL. Esta extensión detectará el `languageserver` y lo lanzará en segundo plano al abrir archivos `.R` o trabajar en un script R, habilitando funcionalidades como completion y hints en el editor.

Plotting: Visor de Gráficas con httpgd

En entornos sin interfaz gráfica (como WSL sin un servidor X11), la visualización de gráficas de R puede lograrse mediante dispositivos gráficos especiales. La extensión de R para VS Code recomienda el paquete **httpgd**, que actúa como un servidor de gráficos vía web sockets. Instálalo igualmente desde R:

```
install.packages("httpgd")
```

`httpgd` provee un dispositivo gráfico que sirve las imágenes (SVG) vía HTTP/WebSocket, permitiendo a VS Code mostrar las gráficas interactivamente ³². Tras instalarlo, abre VS Code, y en la configuración de la extensión R busca la opción de **Gráficos**. Asegúrate que esté configurado para usar **httpgd**. Cuando generes una gráfica en R (por ejemplo con `plot()` o usando `ggplot2`), VS Code abrirá automáticamente un panel con la visualización interactiva de la gráfica.

Consola mejorada: radian (opcional)

Por defecto, la extensión R lanza el intérprete estándar de R en la terminal de VS Code. Opcionalmente, puedes instalar **radian**, una consola alternativa de R escrita en Python, que provee mejoras como autocompletado avanzado, sintaxis resaltada en la consola y mejor manejo de la historia ³³. Para instalar radian, primero instala pip (si no lo tienes) y luego ejecuta:

```
pipx install radian # o pip install --user radian
```

Luego, en la configuración de VS Code, especifica la ruta del ejecutable radian en la opción `r.rterm.windows` o `r.rterm.linux` según corresponda, y habilita `r.bracketedPaste`. Con esto, al iniciar una sesión R en VS Code, utilizará radian proporcionando una experiencia más rica en la consola (por ejemplo, paréntesis coloreados, autocompletar mientras escribes, etc.).

Dependencias de sistema para paquetes de R

Muchos paquetes de R escritos en C/C++ o que envuelven bibliotecas externas requieren tener instaladas ciertas dependencias de sistema. Ya instalamos `r-base-dev` que incluye las herramientas de compilación básicas. Sin embargo, podrías necesitar otros paquetes de desarrollo: - Por ejemplo, para bases de datos o APIs web: `libcurl4-openssl-dev` (para curl), `libxml2-dev` (para XML), `libssl-dev` (para SSL), etc.

- Para ciertos paquetes de gráficos o ciencia de datos: `libpng-dev`, `libjpeg-dev`, `libatlas-base-dev`, `gfortran`, etc.

La necesidad exacta dependerá del paquete R que instales. Si al usar `install.packages()` en R ves errores de compilación indicando falta de alguna librería, busca el nombre de la librería y probablemente instálala con apt ese `-dev`. Por ejemplo, si falla por `curl`, instala `libcurl4-openssl-dev` y vuelve a intentarlo.

En resumen, con R instalado y configurado con **languageserver** para VS Code y **httpgd** para gráficos, tendrás una experiencia de desarrollo cómoda: VS Code te brindará iluminación de sintaxis, autocompletado y ayuda en tus scripts R ³¹, y podrás ejecutar código R directamente en la terminal integrada o en notebooks (R Markdown/Quarto) con visualización de resultados y gráficas dentro del editor.

Integración de Quarto

Quarto es una plataforma unificada para documentos reproducibles, que permite combinar Markdown con código ejecutable de R, Python, Julia y otros lenguajes en un mismo documento. Es la evolución de R Markdown, desarrollada por Posit (antes RStudio), orientada a generar reportes, libros, sitios web o presentaciones de forma reproducible. Integrar Quarto en nuestro entorno nos permitirá crear informes con texto, código y gráficas de forma integrada, aprovechando tanto R como Python en un solo documento `.qmd`.

Instalación de la CLI de Quarto en WSL

Para poder renderizar documentos Quarto, necesitamos instalar la herramienta de línea de comando de Quarto en Ubuntu WSL. Posit distribuye instaladores para Linux en formato `.deb`:

1. Ve a la página oficial de Quarto (quarto.org) y descarga el paquete `.deb` para Ubuntu (por ejemplo, para `x86_64`). También puedes copiar el enlace de descarga.
2. En la terminal de Ubuntu, usa `wget` para descargar el `.deb`. Por ejemplo:

```
wget https://github.com/quarto-dev/quarto-cli/releases/download/v1.8.9/quarto-1.8.9-linux-amd64.deb -O quarto.deb
```

(Nota: La versión y URL exacta pueden cambiar, verifica la última versión estable.)

3. Instala el paquete con `dpkg`:

```
sudo dpkg -i quarto.deb
```

Si ves errores de dependencias, ejecuta `sudo apt-get install -f` para instalarlas automáticamente ³⁴.

4. Verifica la instalación con: `quarto --version`. Deberías obtener el número de versión de Quarto instalado.

Uso de Quarto con VS Code

Ya habíamos instalado la extensión **Quarto** en VS Code (si no, instálala desde el Marketplace). Esta extensión proporciona integración completa: comandos para renderizar documentos, atajos de teclado, realce de sintaxis, autocompletado de opciones YAML y de código embebido (R/Python), vista previa en vivo de los documentos, etc ³⁵.

Con Quarto instalado, puedes crear un nuevo documento Quarto (archivo `.qmd`). Por ejemplo, crea `analysis.qmd` con contenido R y Python mezclado en diferentes celdas. En VS Code, puedes renderizar el documento de varias formas:

- Abre el comando **"Quarto: Render"** o **"Quarto: Preview"** desde la paleta de comandos (Ctrl+Shift+P) o usa el atajo predeterminado (Ctrl+Shift+K abre la previsualización). La extensión ejecutará `quarto preview`, que renderiza el documento y abre una vista previa interactiva en VS Code, alineada junto al editor ³⁶. Cada vez que guardes el documento, Quarto recargará la vista previa con los últimos resultados, incluyendo las salidas y gráficas generadas por el código.
- Puedes cambiar el formato de salida (HTML por defecto) a PDF, Word, etc., ajustando el YAML del encabezado del documento o usando el comando **"Quarto: Preview Format"** que permite elegir el formato a previsualizar.

Quarto se integra con los kernels de Jupyter y con R directamente, por lo que al renderizar ejecutará el código Python con Python (o Jupyter) y el código R con R, incrustando los resultados en el documento final. Puedes configurar qué kernel o intérprete usar para cada lenguaje en el YAML del documento (por defecto usará el R instalado en WSL para chunks `{r}` y el Python actual para chunks `{python}`).

Al utilizar Quarto en VS Code obtienes una poderosa combinación: - Puedes escribir informes mezclando texto, LaTeX, Markdown y código de múltiples lenguajes. - Obtienes **vista previa en vivo** dentro de VS Code de los resultados formateados ³⁵ (incluso con interactividad básica en gráficos HTML/JavaScript). - Mantienes todo en un flujo reproducible: Quarto manejará la ejecución del código en orden y la inclusión de resultados, facilitando la colaboración y la verificación de que los resultados corresponden al código fuente.

Por ejemplo, un científico de datos puede crear un documento `analisis.qmd` donde carga datos con Python, entrena un modelo de machine learning, y luego en otra sección usar R para generar visualizaciones avanzadas, todo en el mismo documento. Con Quarto, al renderizar, ambos lenguajes producen una salida unificada en HTML/PDF.

Finalmente, destacar que Quarto también permite generar presentaciones (reveal.js, Beamer), blogs o libros completos, lo que lo hace una herramienta muy versátil en el entorno de data science.

(Si deseas profundizar, consulta la [documentación oficial de Quarto](#) que incluye tutoriales desde "Hello, Quarto" hasta temas avanzados.)

Integración de Docker con WSL 2 y Manejo de Permisos

Docker se ha convertido en una herramienta fundamental para reproducir entornos y desplegar aplicaciones. Gracias a WSL 2, es posible utilizar Docker de forma eficiente en Windows sin necesidad de

máquinas virtuales pesadas, aprovechando el kernel Linux integrado ³⁷. Hay dos formas principales de configurar Docker con WSL:

Opción 1: Docker Desktop (recomendada). Docker Desktop para Windows utiliza WSL 2 como backend, proporcionando una integración fluida. Pasos resumen: - Instala Docker Desktop en Windows (disponible en la web de Docker). Asegúrate de cumplir los requisitos mínimos (Windows 10 21H2+ o Windows 11) y tener WSL 2 habilitado ³⁸. - Inicia Docker Desktop y ve a *Settings > General*: verifica que "Use WSL 2 based engine" esté activado. En *Settings > Resources > WSL Integration*: habilita la integración con tu distribución Ubuntu WSL. - Docker Desktop iniciará el demonio de Docker dentro de WSL2 automáticamente. En tu terminal de Ubuntu WSL, prueba el comando:

```
docker run hello-world
```

Esto descargará y ejecutará el contenedor de prueba "hello-world". Si todo está correctamente configurado, verás un mensaje de éxito desde el contenedor.

Es posible que al ejecutar Docker en WSL por primera vez obtengas un error de permiso denegado al acceder al socket de Docker. Para solucionarlo, añade tu usuario de Linux al grupo `docker` dentro de WSL:

```
sudo usermod -aG docker $USER
```

Luego cierra y vuelve a abrir la sesión (o ejecuta `newgrp docker`). Esto permite usar Docker sin `sudo` ³⁹. Docker Desktop configura automáticamente el socket con este grupo. (En algunos casos, también es necesario agregar tu usuario de Windows al grupo local "docker-users", aunque Docker Desktop suele hacerlo durante la instalación si usas una cuenta de administrador).

Opción 2: Instalación nativa de Docker Engine en WSL. También puedes instalar Docker directamente en Ubuntu WSL (sin Docker Desktop). Este enfoque puede ser útil en entornos de servidor o si prefieres no instalar Docker Desktop. Los pasos generales serían: - Instalar los paquetes de Docker Engine en Ubuntu (por ejemplo, usando el repositorio oficial de Docker CE). Esto típicamente involucra instalar `docker-ce`, `docker-ce-cli` y `containerd.io` con apt. - *Habilitar systemd en WSL*: Docker Engine en Linux normalmente se gestiona con el servicio systemd. WSL 2 ahora soporta systemd (debe habilitarse añadiendo en `/etc/wsl.conf` de la distro:

```
[boot]
systemd=true
```

y reiniciando la distro). Una vez con systemd activo, el demonio de Docker podrá iniciarse automáticamente. - Añadir tu usuario al grupo docker, igual que en pasos anteriores, para evitar usar sudo ³⁹. - Verificar con `docker run hello-world`.

Este método requiere más pasos manuales y mantener Docker actualizado por tu cuenta, pero evita tener el componente Desktop en Windows. Para muchos usuarios de WSL, Docker Desktop simplifica las cosas al encargarse del networking, actualizaciones y compatibilidad.

Uso de Docker en VS Code: Dado que instalamos la extensión *Docker* en VS Code, podrás manejar tus contenedores e imágenes desde la interfaz (barra lateral de Docker) al tener Docker funcionando en WSL. Además, la extensión *Remote - Containers* (incluida en el pack de Remote Development) te permite abrir entornos de desarrollo dentro de contenedores Docker fácilmente (usando archivos `devcontainer.json`), lo cual puede ser útil para aislar completamente tu entorno de desarrollo. Todo esto funciona sin problemas gracias a la compatibilidad de Docker con WSL 2 ³⁷.

Tip de rendimiento: El uso de Docker en WSL 2 es muy eficiente, pero si ejecutas cargas intensivas, considera habilitar la opción experimental **autoMemoryReclaim** de WSL ⁴⁰. Esta característica (disponible desde WSL 1.3.10) permite reclamar memoria no utilizada de la VM de WSL de vuelta a Windows, evitando que, por ejemplo, tras construir imágenes Docker muy grandes la memoria quede asignada a WSL. Puedes habilitarlo añadiendo en tu archivo `.wslconfig` (en Windows, en la ruta `%UserProfile%\wslconfig`):

```
[wsl2]
autoMemoryReclaim=1
```

y reiniciando WSL. Esto ayudará a mantener un uso de RAM más estable durante procesos con contenedores.

En resumen, con Docker integrado en WSL podrás ejecutar contenedores Linux directamente desde la terminal de Ubuntu, compartir volúmenes con el sistema de archivos de Windows y aprovechar la aceleración hardware (e.g. GPU, si aplicable) casi como si estuvieras en Linux nativo.

Flujo de Trabajo Recomendado

Con todas las piezas instaladas, a continuación se describen buenas prácticas para organizar el trabajo diario de desarrollo en Data Science/ML, aprovechando las herramientas configuradas:

Gestión de entornos y dependencias

Para cada proyecto de data science o machine learning, se recomienda crear un entorno virtual aislado. Utiliza **Poetry** o **uv** (o en su defecto `venv` manualmente) para inicializar un nuevo proyecto con su propio entorno. Esto garantiza que las dependencias de un proyecto no interfieran con las de otro, y facilita la reproducibilidad. Al empezar un proyecto: - Inicia un repositorio git nuevo (por ejemplo con `git init` o `gh repo create` si usas GitHub). - Crea el entorno: si usas Poetry, `poetry init` y `poetry install` crearán un `venv` con un `pyproject.toml` inicial; si usas uv, `uv init` hará similar. Define allí todas las dependencias requeridas (y marca las de desarrollo, como las herramientas de formateo/lint, en el grupo adecuado). - Fija versiones para producción: asegúrate de mantener actualizado el lockfile (e.g. `poetry.lock` o `uv-lock.json`) y versionarlo en git para que otros puedan instalar exactamente las mismas versiones.

En proyectos R, aunque R no tiene entornos virtuales análogos a Python, puedes lograr aislamiento usando **{renv}** (un paquete R para gestionar librerías por proyecto) o simplemente documentando la sesión y usando contenedores Docker/renv para aislar. Si trabajas interactivamente con R en VS Code, las dependencias R instaladas irán a la librería global de la distro (o a una librería específica de usuario).

Estructura de proyectos

Adoptar una estructura de archivos coherente facilita la colaboración y la mantenibilidad. Por ejemplo, el patrón de **Cookiecutter Data Science** ⁴¹ sugiere:

```
├─ README.md          <- Descripción del proyecto, instrucciones.
├─ data/              <- Datos crudos, subdir para datos procesados, etc.
├─ notebooks/         <- Notebooks Jupyter (o Quarto) exploratorios.
├─ src/               <- Código fuente del proyecto (módulos, scripts).
├─ tests/             <- Tests automatizados (si aplican).
├─ reports/           <- Reportes generados (HTML, PDF, etc. por Quarto).
├─ pyproject.toml     <- Especificación de dependencias (Poetry/uv) para
Python.
└─ ...                <- Otros archivos de config (Dockerfile, .gitignore,
etc.)
```

Mantén los datos (especialmente los voluminosos) fuera del control de versiones, idealmente en la carpeta `data/` que agregues a `.gitignore`. Los notebooks y reportes generados tampoco deberían versionarse una vez finalizados (en su lugar, conserva el código fuente para generarlos). Si usas Quarto, versiona los `.qmd` pero no necesariamente los HTML/PDF resultantes (pueden ser artefactos de release).

Control de versiones con Git y GitFlow

Usa Git desde el inicio del proyecto. Commits frecuentes y mensajes descriptivos ayudan a rastrear cambios. Es buena idea establecer un flujo de branching profesional; **GitFlow** es una opción clásica ⁴². En GitFlow: - El repositorio tiene dos ramas principales de larga vida: `main` (o `master`) que contiene las versiones en producción o liberadas, y `develop` que acumula el trabajo integrando features para la siguiente versión. - Para cada nueva funcionalidad o experimento, crea una rama a partir de `develop` (por ejemplo `feature/nueva-analitica`). Allí desarrollas aisladamente. Cuando esté lista, se fusiona de vuelta a `develop` (mediante un Pull Request, idealmente, para revisión de código). - Periódicamente, cuando `develop` ha acumulado suficientes cambios estables, se crea una rama `release/x.y.z` para preparar una versión; al finalizar, se fusiona a `main` (y taguea una versión) y también esos cambios a `develop` si correspondiera. - Si surge un bug urgente en producción, se crea una rama desde `main` llamada `hotfix/bug` para corregirlo, y se fusiona de vuelta a `main` y `develop`.

Este modelo mantiene el código organizado y facilita colaborar en equipo. No obstante, GitFlow puede ser excesivo para proyectos pequeños; en esos casos, un flujo más simple (tipo GitHub Flow: ramas cortas desde `main` directamente) podría bastar. En todo caso, aprovecha las herramientas de Git integradas en VS Code: el panel de control de versiones te permite ver cambios, hacer stage/commit, y las extensiones pueden ejecutarse hooks (como pre-commit) al dar commit para garantizar calidad.

Recuerda agregar un archivo `.gitignore` apropiado: incluye `*.env` (credenciales), archivos de configuración local, la carpeta de entornos virtuales (si Poetry/uv la crean dentro del proyecto), datos, outputs, etc., para no subirlos al repositorio.

Pruebas (Testing)

Incluir tests automáticos desde etapas tempranas mejora la confiabilidad del código: - En Python, utiliza **pytest** (u otra framework) para escribir pruebas unitarias para las funciones críticas de tu proyecto. Ubícalas en la carpeta `tests/` y nomenclatura como `test_*.py`. Puedes ejecutar `pytest` directamente en VS Code (la extensión de Python detecta tests) o integrarlo con pre-commit (existe un hook para pytest) o en pipelines de CI. - En R, si estás desarrollando un paquete o análisis complejo, considera usar **{testthat}** para estructurar pruebas de unidades. Aunque en trabajos exploratorios de data science no siempre se escriben tests formales, es buena práctica testear las funciones de transformación de datos o cálculos importantes. - Realiza también *tests manuales exploratorios*: por ejemplo, verifica que un DataFrame tiene las columnas esperadas tras una operación, o que una visualización no tiene valores atípicos evidentes. Este tipo de comprobaciones suelen hacerse en notebooks durante la exploración, pero puedes convertirlas en aserciones dentro de tus scripts para ejecutarlas rutinariamente.

Incorporar tests en el flujo de desarrollo (por ejemplo, ejecutando `pytest` antes de cada commit o al menos en cada push en un sistema CI) previene que errores regresen inadvertidamente cuando el código crece.

Notebooks y Visualización

Los **notebooks** son fundamentales en data science para la exploración y comunicación. En este entorno: - Puedes utilizar Jupyter notebooks (.ipynb) directamente en VS Code: la extensión de Python proporciona una experiencia interactiva donde puedes ejecutar celdas, ver gráficos inline, etc. Incluso puedes "conectarte" a tu entorno WSL para que los notebooks se ejecuten allí. - Una alternativa potente es usar **Quarto** con notebooks literarios (.qmd), como describimos antes, especialmente para reportes reproducibles que combinen texto y código de R/Python. - Mantén los notebooks organizados en la carpeta `notebooks/`, y procura limpiarlos (eliminar resultados o outputs pesados) antes de subirlos a git para evitar diffs ruidosos. Puedes usar pre-commit con hooks como `nbstripout` o `nb-clean` para esto.

En cuanto a **visualización**, aprovecha tanto las bibliotecas de Python (matplotlib, seaborn, plotly, bokeh) como las de R (ggplot2, lattice, etc.) según te convenga: - VS Code mostrará las gráficas generadas en notebooks o en scripts interactivos (por ejemplo, si usas la opción "Run Cell" `#%%` en un script Python, la salida gráfica aparece en el **Interactive Window** de VS Code). - Con R, gracias a `httpgd`, las gráficas aparecen en el panel de Plots de VS Code. Incluso puedes copiarlas o guardarlas desde allí. - Para trabajos colaborativos, considera exportar visualizaciones a formatos estáticos (PNG, SVG) o interactivos (HTML) que puedan ser versionados o incluidos en informes. - Si necesitas dashboards, puedes integrar herramientas como **Plotly Dash**, **Streamlit** o **Shiny** (para R) en tu flujo, ejecutándolos dentro de WSL y accediendo vía localhost en el navegador de Windows.

Por último, adopta la costumbre de documentar tu trabajo: añade descripciones en notebooks, comentarios en el código, y quizás utiliza **README.md** o incluso documentos Quarto para explicar resultados. Esto convierte tus hallazgos en reportes comprensibles para otros.

Siguiendo esta guía, un profesional técnico podrá reproducir un entorno robusto de desarrollo en Windows con WSL2, VS Code, Python y R, combinando lo mejor de ambos ecosistemas. La configuración aquí descrita proporciona aislamiento, reproducibilidad y herramientas de calidad, cubriendo desde la instalación hasta el flujo de trabajo diario. ¡Happy coding!

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
30 31 32 33 28 29 35 36 37 38 39 40 41 42

1 3 4 5 7 **Instalación de WSL | Microsoft Learn**

<https://learn.microsoft.com/es-es/windows/wsl/install>

2 6 **Guía para Activar WSL 2 en Windows - Binario 0**

<https://binariocero.com/windows/guia-para-activar-wsl-2-en-windows>

8 9 11 12 13 14 15 16 **Get started using VS Code with WSL | Microsoft Learn**

<https://learn.microsoft.com/en-us/windows/wsl/tutorials/wsl-vscode>

10 **Remote development in WSL**

<https://code.visualstudio.com/docs/remote/wsl-tutorial>

17 **GitHub - pyenv/pyenv: Simple Python version management**

<https://github.com/pyenv/pyenv>

18 19 **pipx**

<https://pipx.pypa.io/>

20 21 22 25 26 **Loopwerk: Poetry versus uv**

<https://www.loopwerk.io/articles/2024/python-poetry-vs-uv/>

23 **uv**

<https://docs.astral.sh/uv/>

24 **Installation | uv - Astral Docs**

<https://docs.astral.sh/uv/getting-started/installation/>

27 **pre-commit**

<https://pre-commit.com/>

28 29 **Ubuntu Packages For R - Full Instructions**

<https://cran.r-project.org/bin/linux/ubuntu/fullREADME.html>

30 31 32 33 **R in Visual Studio Code**

<https://code.visualstudio.com/docs/languages/r>

34 **ψML - Getting Quarto Running from WSL**

https://simonstolarczyk.com/posts/misc/quarto_preview.html

35 36 **VS Code – Quarto**

<https://quarto.org/docs/tools/vscode.html>

37 38 40 **WSL | Docker Docs**

<https://docs.docker.com/desktop/features/wsl/>

39 Post-installation steps | Docker Docs

<https://docs.docker.com/engine/install/linux-postinstall/>

41 Cookiecutter Data Science

<https://cookiecutter-data-science.drivendata.org/>

42 Flujo de trabajo de Gitflow | Atlassian Git Tutorial

<https://www.atlassian.com/es/git/tutorials/comparing-workflows/gitflow-workflow>