INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE MONTERREY

MNA - Maestría en Inteligencia Artificial Aplicada

Fase 2 | Avance de Proyecto

A01224696 Luis Daniel Castillo Alegría
A01795214 Raúl Eduardo Gómez Godínez
A01796393 Fabiola Guevara Soriano
A01796404 Manuel Alejandro Vázquez Meza
A01795907 Nancy Teresa Zapién García

Docente Titular:
Dr. Gerardo Rodríguez Hernández
Mtro. Ricardo Valdez Hernández

Docente Asistente:
Mtra. María Mylen Treviño Elizondo

Docente Tutor:
Guillermina Rivera Hernández

Operaciones de Aprendizaje Automático
02 de noviembre 2025

TABLA DE CONTENIDO

1. EQUIPO MLOPS

DevOps

A01796404

Manuel Alejandro Vázquez Meza

Data Enginer

A01795907

Nancy Teresa Zapién García

ML Engineer

A01795214

Raul Eduardo Gomez Godinez

Software Engineer

A01796393

Fabiola Guevara Soriano

Data Scientist

A01224696

Luis Daniel Castillo Alegría

2. INTRODUCCIÓN AL PROYECTO

En esta segunda fase del proyecto se consolida el desarrollo iniciado en la Fase 1, avanzando desde el análisis del problema y la propuesta de valor hacia la implementación estructurada, escalable y profesional de una solución de Machine Learning. El objetivo principal de esta etapa es demostrar la capacidad de aplicar principios de ingeniería de software y prácticas de MLOps que garanticen la reproducibilidad, mantenibilidad y trazabilidad de todo el ciclo de vida del modelo.

Esta fase se centra en la gestión integral del proyecto de Machine Learning, reforzando la transición desde un prototipo exploratorio hacia un sistema sólido y gestionado, listo para su integración en entornos productivos. A lo largo de las siguientes actividades, se abordarán los pilares fundamentales que aseguran la calidad técnica y operativa de la solución propuesta:

Estructuración del Proyecto con Cookiecutter
Se implementará una estructura estandarizada basada en Cookiecutter Data Science, con el fin de mantener una organización clara de directorios, módulos y flujos de trabajo. Esta práctica promueve la colaboración entre equipos, facilita la escalabilidad del código y sienta las bases para un ciclo de desarrollo reproducible y sostenible.

Refactorización y modularización del código
El código existente será reorganizado en módulos y clases con responsabilidades definidas, aplicando principios de diseño limpio y programación orientada a objetos (POO). Esta tarea busca mejorar la eficiencia, legibilidad y mantenibilidad, garantizando que las futuras iteraciones y actualizaciones del modelo puedan realizarse sin comprometer la integridad del sistema.

Implementación de un pipeline de modelado con Scikit-Learn
Se desarrollará un pipeline automatizado que integre las etapas de preprocesamiento, entrenamiento y evaluación del modelo. Este enfoque asegura la consistencia en las ejecuciones, facilita la validación cruzada de resultados y permite documentar cada componente del flujo de trabajo de forma clara y reproducible, siguiendo las mejores prácticas de la industria.

Seguimiento de experimentos y gestión de modelos
Se incorporarán herramientas como MLflow y DVC para el registro, versionado y monitoreo de los experimentos. Cada ejecución será documentada con sus hiperparámetros, métricas de desempeño, versiones de modelo y configuraciones específicas. Además, se emplearán las funcionalidades de visualización de MLflow para presentar los resultados de forma comprensible y comparativa, fortaleciendo el proceso de toma de decisiones y la trazabilidad del desarrollo.

En conjunto, estas actividades consolidan el proyecto dentro de un entorno MLOps robusto, garantizando que el modelo de predicción de demanda de bicicletas no solo sea preciso, sino también reproducible, versionado y escalable. La Fase 2, por tanto, marca el punto de madurez técnica del proyecto, integrando buenas prácticas de ingeniería de datos y ciencia aplicada para construir una solución confiable que aporte valor sostenible a la movilidad urbana.

3. ESTRUCTURACIÓN DEL PROYECTO CON COOKIECUTTER

3.1. Justificación y Objetivo

La Fase 1 del proyecto se centró en la exploración de datos y el desarrollo de un modelo baseline, concentrando la lógica principalmente en un Jupyter Notebook (Seoul_Bike_EDA_ML.ipynb). Si bien este enfoque es ideal para la experimentación inicial, presenta desafíos de mantenibilidad, reproducibilidad y colaboración a largo plazo.

Para abordar estos desafios y alinear el proyecto con las mejores prácticas de la industria, el primer paso de la Fase 2 fue adoptar una estructura de proyecto estandarizada. El objetivo es transformar el prototipo exploratorio en una base de código robusta, modular y escalable.

3.2. Implementación con Cookiecutter

Se utilizó la plantilla cookiecutter-data-science como base para reorganizar el proyecto. Esta herramienta automatiza la creación de una estructura de directorios y archivos lógica y predefinida, diseñada específicamente para proyectos de Machine Learning. La adopción de esta plantilla nos proporciona los siguientes beneficios clave:

Claridad y organización: Separa de forma lógica los datos, el código fuente, los notebooks, los modelos y los reportes.

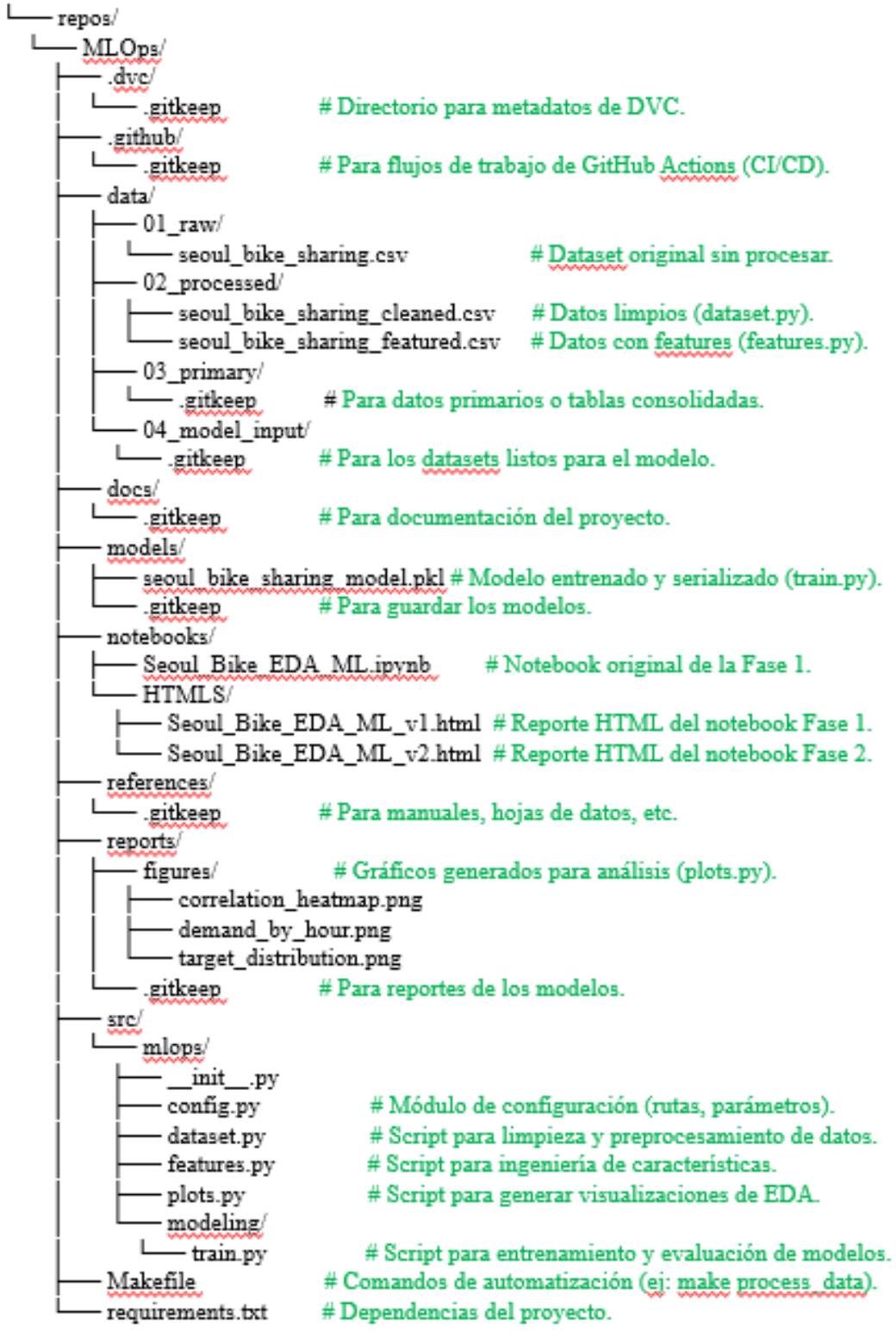
Reproducibilidad: Facilita que otros miembros del equipo e incluso nosotros mismos en el futuro (Equipo 44) puedan comprender y ejecutar el proyecto sin ambigüedades.

Escalabilidad: Permite que el proyecto crezca de manera ordenada, añadiendo nuevas funcionalidades o modelos sin comprometer la estructura existente.

Colaboración: Estandariza la ubicación de los artefactos del proyecto, haciendo que la colaboración en equipo sea más eficiente.

3.3. Nueva Estructura del Proyecto

El código y los artefactos generados en la Fase 1 se migraron a la nueva estructura de directorios. A continuación, se describe la organización resultante y el propósito de cada componente principal:



Esta nueva organización representa un salto cualitativo desde el notebook monolítico, sentando las bases para las siguientes etapas de refactorización de código, implementación de pipelines y seguimiento de experimentos.

4. ESTRUCTURACIÓN Y REFACTORIZACIÓN DEL CÓDIGO

4.1. Justificación y objetivo

El notebook de la Fase 1 (Seoul_Bike_EDA_ML.ipynb) fue fundamental para la exploración inicial y la creación de un modelo baseline. Sin embargo, su naturaleza monolítica (mezclando carga de datos, limpieza, visualización y modelado en un solo flujo secuencial) lo hace inadecuado para un entorno de producción por las siguientes razones:

- Baja mantenibilidad: Un cambio en una etapa (ej. limpieza de datos) podría requerir la re-ejecución de todo el notebook y tener efectos inesperados en celdas posteriores.
- Dificultad para reutilizar: La lógica no está encapsulada, lo que dificulta reutilizar funciones de preprocesamiento o entrenamiento en otros contextos.
- Complejidad para pruebas: Es muy difícil realizar pruebas unitarias a celdas individuales de un notebook.
- Legibilidad reducida: A medida que el proyecto crece, el notebook se vuelve largo y difícil de seguir.

El objetivo de esta fase fue refactorizar el código, migrando la lógica del notebook a una estructura de módulos de Python (.py) bien definidos y organizados, aplicando principios de Programación Orientada a Objetos (POO).

4.2. Proceso de refactorización

El proceso consistió en descomponer el flujo del notebook en responsabilidades lógicas y encapsular cada una en su propia clase y módulo:

1. Modularización: Se crearon scripts de Python dentro del directorio src/mlops/ para cada una de las principales etapas del pipeline:
    - dataset.py: Para la carga y limpieza inicial de los datos.
    - features.py: Para la ingeniería y creación de nuevas características.
    - plots.py: Para la generación de gráficos de análisis exploratorio.
    - modeling/train.py: Para el entrenamiento, ajuste y evaluación de modelos.

2. Aplicación de POO: En lugar de usar funciones sueltas, se implementaron clases para gestionar el estado y la lógica de cada etapa. Esto mejora la cohesión y reduce el acoplamiento.
    - DatasetProcessor (en dataset.py): Encapsula todo lo relacionado con la carga y limpieza del dataset crudo.
    - FeatureEngineer (en features.py): Contiene los métodos para crear características temporales, cíclicas y de interacción.
    - PlotGenerator (en plots.py): Organiza la lógica para generar y guardar las visualizaciones del EDA.
    - ModelTrainer (en modeling/train.py): Abstrae todo el pipeline de entrenamiento, desde la carga de datos procesados hasta la evaluación y guardado del modelo.

3. Centralización de la Configuración: Se creó un archivo config.py para almacenar constantes como rutas de directorios (raw_data_dir, processed_data_dir), nombres de columnas (target_col) y parámetros por defecto. Esto desacopla la configuración de la lógica del código, permitiendo modificar parámetros fácilmente sin tocar los scripts principales.

4.3. Ejemplo de refactorización

Ejemplo de refactorización de una celda a método de clase. Para ilustrar el cambio, una secuencia de celdas en el notebook que realizaba la limpieza de datos:

```
# En el Notebook (Fase 1)
df = pd.read_csv('...')
df.columns = df.columns.str.lower().str.replace(' ', '_')
df = df[df['functioning_day'] == 'Yes']
df.to_csv('cleaned.csv')
```

Se transformó en una clase cohesiva y reutilizable en dataset.py:

```
# En dataset.py (Fase 2)
class DatasetProcessor:
    def __init__(self, input_path, output_path):
        # ... inicialización ...

    def load_data(self):
        # ...
        return self

    def preprocess_data(self):
        self._normalize_column_names()
        self.df = self.df[self.df["functioning_day"] ==
"Yes"].copy()
        # ... más lógica ...
        return self

    def save_data(self):
        # ...
        pass

# Uso:
processor = DatasetProcessor('raw.csv', 'cleaned.csv')
processor.load_data().preprocess_data().save_data()
```

4.4. Importancia y beneficios obtenidos

Esta refactorización es un pilar fundamental para un proyecto de MLOps profesional. Los beneficios directos son:

- Mantenibilidad: El código es más fácil de entender, depurar y modificar.
- Testabilidad: Cada clase y método puede ser probado de forma aislada (unit testing).
- Reusabilidad: Las clases como DatasetProcessor o ModelTrainer pueden ser importadas y utilizadas en otros pipelines o scripts.
- Escalabilidad: La estructura modular permite añadir nuevas funcionalidades (ej. un nuevo tipo de modelo o paso de preprocesamiento) sin afectar el código existente.

Con esta base de código estructurada, nuestro proyecto está listo para implementar un pipeline automatizado y el seguimiento de experimentos.

5. APLICACIÓN DE MEJORES PRÁCTICAS DE CODIFICACIÓN EN EL PIPELINE DE MODELADO

5.1. Justificación y Objetivo

En las fases anteriores, el preprocesamiento (imputación, codificación de categóricas) y el entrenamiento del modelo se realizaban como pasos secuenciales y manuales. Este enfoque, aunque funcional, es propenso a errores, especialmente a la fuga de datos (data leakage). La fuga de datos ocurre cuando información del conjunto de prueba (o validación) se "filtra" accidentalmente en el proceso de entrenamiento, llevando a métricas de rendimiento infladas y poco realistas.

El objetivo de esta etapa fue implementar un scikit-learn Pipeline para encapsular y automatizar toda la secuencia de preprocesamiento y modelado. Esta es una práctica estándar en la industria que garantiza la robustez y reproducibilidad del flujo de trabajo.

5.2. Implementación del Pipeline en train.py

Dentro de la clase ModelTrainer, se implementó el método _build_pipeline para construir dinámicamente el pipeline de modelado. Este pipeline integra todos los pasos necesarios, desde la transformación de los datos de entrada hasta la instancia del modelo a entrenar.

La estructura del pipeline es la siguiente:

ColumnTransformer para preprocesamiento diferenciado: Se utiliza un ColumnTransformer para aplicar transformaciones específicas a diferentes tipos de columnas, asegurando que cada subconjunto de datos reciba el tratamiento adecuado.

Para columnas numéricas (*num_cols*): Se aplica un SimpleImputer con estrategia de median. La mediana es una elección robusta para imputar valores faltantes, ya que es menos sensible a valores atípicos (outliers) que la media.

Para columnas categóricas (*cat_cols*): Se aplica un OneHotEncoder. Este transformador convierte cada categoría en una nueva columna binaria (0 o 1). Se configuró con handle_unknown='ignore' para que, durante la predicción, si aparece una categoría no vista en el entrenamiento, no genere un error y simplemente la ignore (todas las nuevas columnas de esa feature serán cero).

Instancia del modelo (*model*): El último paso del pipeline es el estimador (el modelo de Machine Learning). El script train.py utiliza un registro (model_registry) para seleccionar dinámicamente el modelo a entrenar (ej. HistGradientBoostingRegressor, LinearRegression, etc.), lo que hace el pipeline flexible y fácil de extender.

El código que define esta estructura en MLOps\mlops\modeling\train.py es:

```
# Dentro de la clase ModelTrainer, método _build_pipeline

# ... (detección de columnas numéricas y categóricas) ...

preprocessor = ColumnTransformer(
    transformers=[
        ("num", SimpleImputer(strategy="median"), num_cols),
        ("cat", OneHotEncoder(handle_unknown="ignore",
sparse_output=False), cat_cols),
    ],
    remainder="drop",
)

return Pipeline(steps=[("preprocessor", preprocessor), ("model",
model_instance)])
```

5.3. Integración con búsqueda de hiperparámetros

La verdadera potencia de esta implementación se manifiesta al integrar el Pipeline con la validación cruzada (TimeSeriesSplit) y la búsqueda de hiperparámetros (GridSearchCV).

El objeto Pipeline completo (preprocesamiento + modelo) se pasa como el estimador a GridSearchCV. Esto garantiza que para cada "fold" de la validación cruzada:

1. Los pasos de preprocesamiento (imputación y codificación) se "aprenden" únicamente con los datos de entrenamiento de ese fold.
2. Luego, se aplican esas transformaciones aprendidas al conjunto de validación de ese mismo fold.
3. Finalmente, se entrena y evalúa el modelo.

Este proceso previene rigurosamente la fuga de datos, ya que el modelo nunca tiene acceso a información del futuro o del conjunto de validación durante su fase de ajuste.

5.4. Importancia y Beneficios

La implementación de scikit-learn Pipelines aporta beneficios cruciales al proyecto:

- Prevenir fuga de datos: Es el beneficio más importante, asegurando que las métricas de evaluación del modelo sean confiables y representativas de su rendimiento en datos nuevos.
- Automatizar y simplificar: El código se vuelve más limpio y el flujo de trabajo más simple. En lugar de llamar a fit, transform y predict en múltiples pasos, solo se necesita llamar a fit y predict en el objeto Pipeline una vez.
- Reproducibilidad: El pipeline completo se puede guardar (serializar) como un único objeto, garantizando que el mismo preprocesamiento y modelo se utilicen de manera consistente tanto en el entrenamiento como en la inferencia.
- Mantenibilidad: Facilita la adición o modificación de pasos de preprocesamiento sin tener que reescribir la lógica de entrenamiento y evaluación.

Con un pipeline robusto y automatizado, el proyecto está listo para la fase final de seguimiento de experimentos y gestión de modelos.

6. SEGUIMIENTO DE EXPERIMENTOS, VISUALIZACIÓN DE RESULTADOS Y GESTIÓN DE MODELOS

6.1. Justificación y Objetivo

A medida que un proyecto de Machine Learning madura, la cantidad de experimentos (diferentes modelos, hiperparámetros, conjuntos de características) crece exponencialmente. Gestionar estos experimentos de forma manual es insostenible y propenso a errores. Es fácil perder la pista de qué combinación de parámetros produjo el mejor resultado o cómo replicar una ejecución específica.

El objetivo de esta etapa fue integrar una herramienta de seguimiento de experimentos para registrar, comparar y gestionar de manera sistemática cada ejecución del pipeline de entrenamiento. Se eligió MLflow por su simplicidad, su potente interfaz de usuario y su capacidad para integrarse de forma nativa con scikit-learn.

La integración de MLflow nos permite:

- Registrar parámetros: Guardar automáticamente los hiperparámetros utilizados en cada ejecución de GridSearchCV.
- Versionar métricas: Almacenar las métricas de rendimiento (como R^2 y RMSE) tanto de la validación cruzada como del conjunto de prueba.

- Gestionar modelos: Guardar el pipeline del modelo serializado como un "artefacto" asociado a su ejecución, asegurando la trazabilidad.
- Visualizar resultados: Utilizar la UI de MLflow para comparar gráficamente el rendimiento de diferentes modelos y configuraciones, facilitando la toma de decisiones.

6.2. Integración de MLflow en el Pipeline de entrenamiento

Para implementar el seguimiento, se modificó el script src/mlops/modeling/train.py. La lógica de MLflow se encapsuló dentro de la clase ModelTrainer, específicamente en los métodos run y _evaluate_and_save.

El flujo de trabajo ahora es el siguiente:

1. Inicio de la ejecución: Antes de comenzar la búsqueda de hiperparámetros, se inicia una nueva ejecución de MLflow con mlflow.start_run().

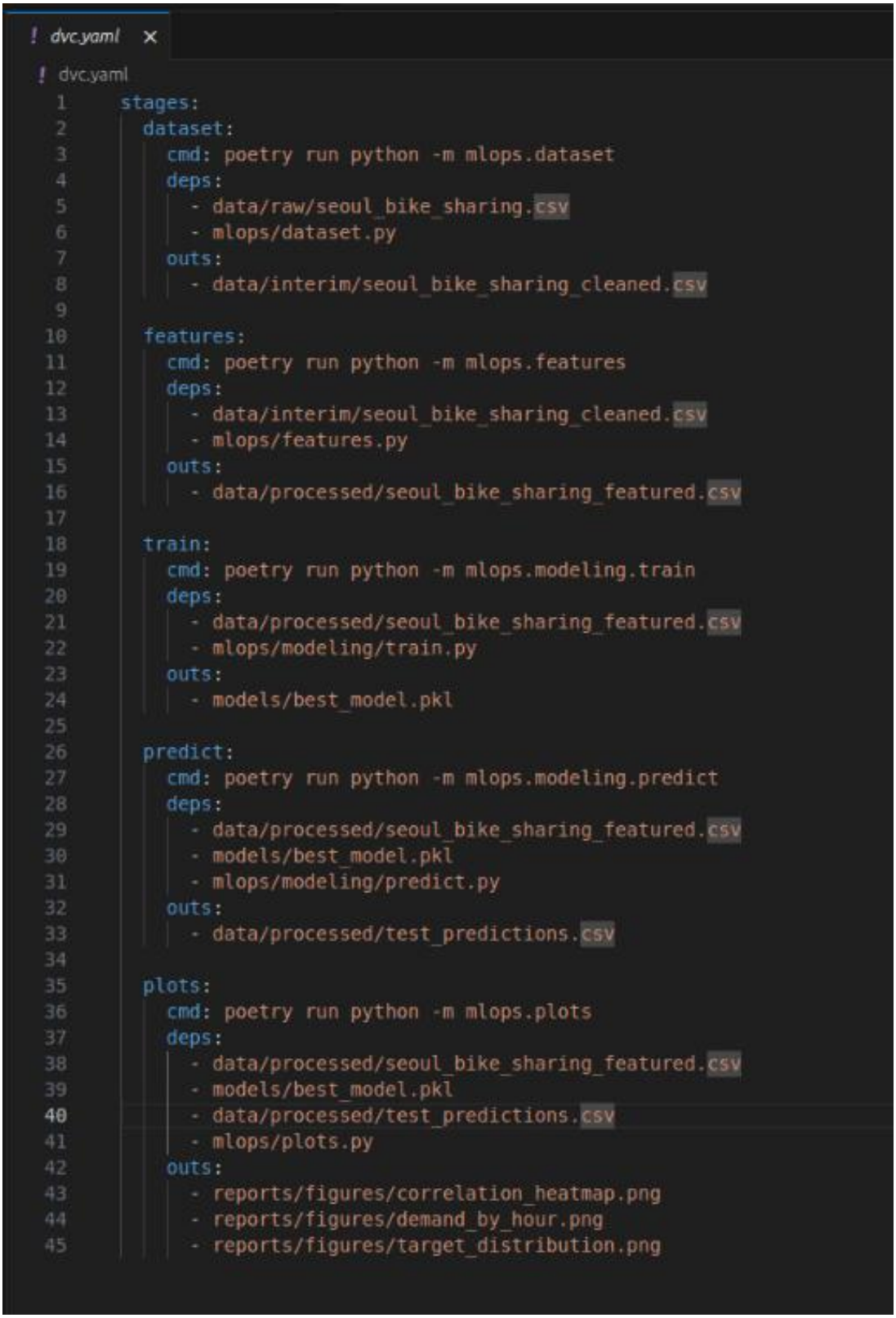
2. Registro de parámetros (log_params): Se registran los parámetros clave del experimento, como el nombre del modelo, la estrategia de búsqueda (GridSearchCV) y la métrica de optimización.

3. Registro de métricas (log_metrics): Una vez que la búsqueda de hiperparámetros finaliza, se registran las métricas más importantes:
    - La mejor puntuación obtenida durante la validación cruzada (best_cv_score).
    - Las métricas finales de R^2 y RMSE calculadas sobre el conjunto de prueba.

4. Registro del modelo (sklearn.log_model): El mejor Pipeline de scikit-learn encontrado por GridSearchCV se guarda directamente en el registro de MLflow. Esto no solo almacena el modelo (.pkl), sino también su conda.yaml y requirements.txt, garantizando una reproducibilidad total.

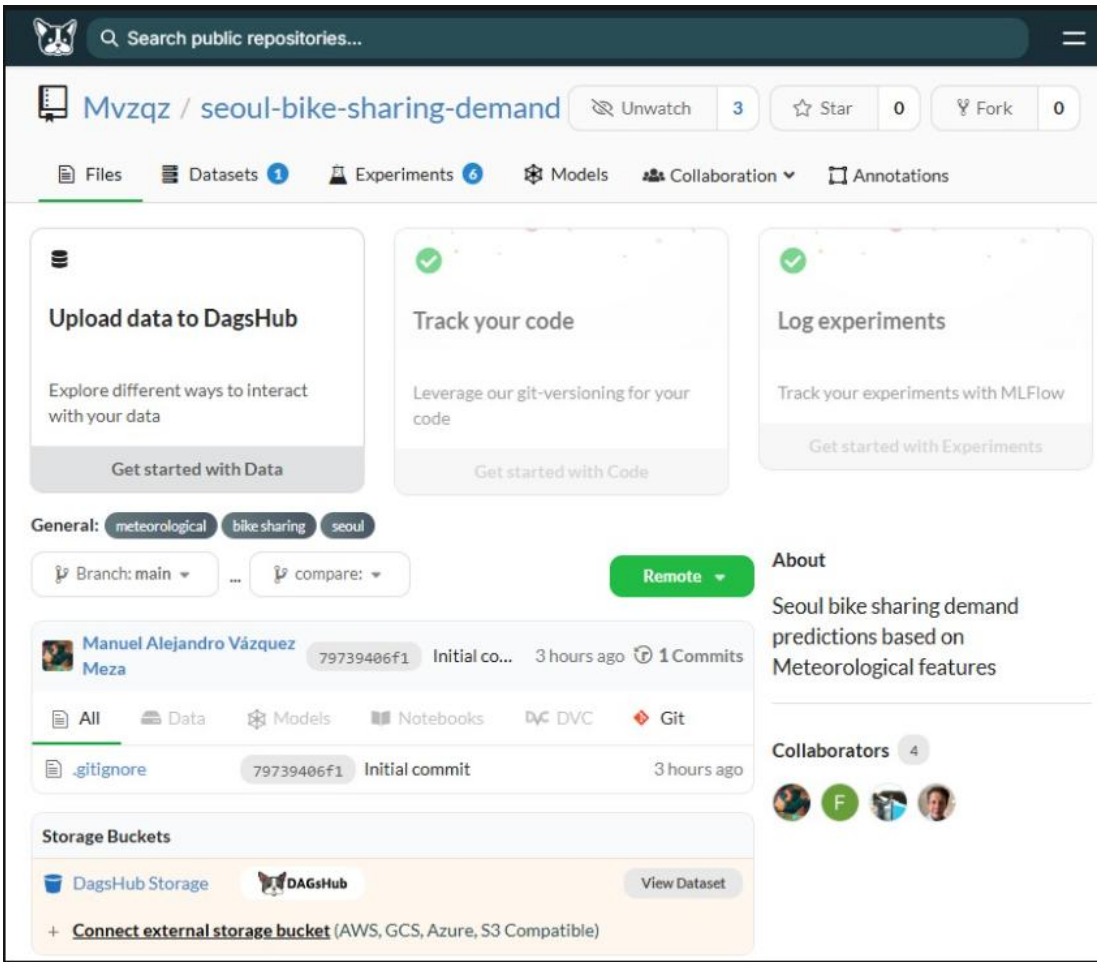
5. Visualización: Una vez ejecutado el script, se puede lanzar la interfaz de MLflow (mlflow ui) para visualizar y comparar todas las ejecuciones, sus parámetros, métricas y modelos asociados.

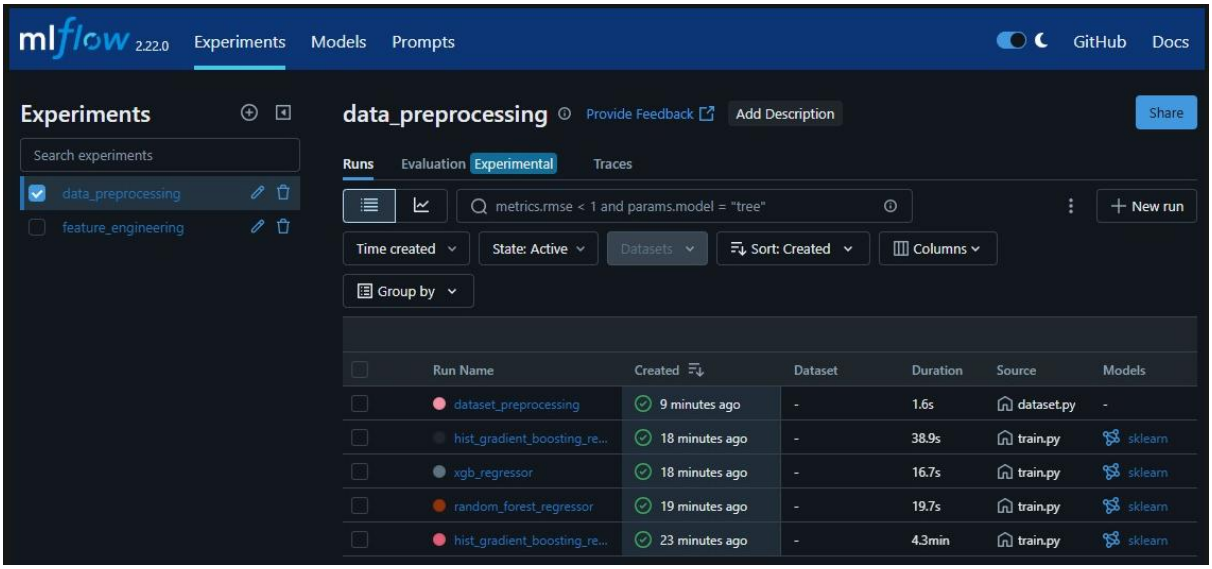
6.3 Configuración de MLflow y DVC



```yaml
stages:
  dataset:
    cmd: poetry run python -m mlops.dataset
    deps:
      - data/raw/seoul_bike_sharing.csv
      - mlops/dataset.py
    outs:
      - data/interim/seoul_bike_sharing_cleaned.csv

  features:
    cmd: poetry run python -m mlops.features
    deps:
      - data/interim/seoul_bike_sharing_cleaned.csv
      - mlops/features.py
    outs:
      - data/processed/seoul_bike_sharing_featured.csv

  train:
    cmd: poetry run python -m mlops.modeling.train
    deps:
      - data/processed/seoul_bike_sharing_featured.csv
      - mlops/modeling/train.py
    outs:
      - models/best_model.pkl

  predict:
    cmd: poetry run python -m mlops.modeling.predict
    deps:
      - data/processed/seoul_bike_sharing_featured.csv
      - models/best_model.pkl
      - mlops/modeling/predict.py
    outs:
      - data/processed/test_predictions.csv

  plots:
    cmd: poetry run python -m mlops.plots
    deps:
      - data/processed/seoul_bike_sharing_featured.csv
      - models/best_model.pkl
      - data/processed/test_predictions.csv
      - mlops/plots.py
    outs:
      - reports/figures/correlation_heatmap.png
      - reports/figures/demand_by_hour.png
      - reports/figures/target_distribution.png
```


manuel-vazquez@manuel-vazquez-KLVD-WXX9:~/Documents/MNA/T4 - MLOPS/MLOps$ poetry run dvc dag
dataset
features
train
predict
plots
manuel-vazquez@manuel-vazquez-KLVD-WXX9:~/Documents/MNA/T4 - MLOPS/MLOps$


Search public repositories...
Mvzqz / seoul-bike-sharing-demand
Unwatch 3
Star 0
Fork 0
Files
Datasets 1
Experiments 6
Models
Collaboration
Annotations
Upload data to DagsHub
Explore different ways to interact with your data
Get started with Data
Track your code
Leverage our git-versioning for your code
Get started with Code
Log experiments
Track your experiments with MLFlow
Get started with Experiments
General: meteorological bike sharing seoul
Branch: main
compare:
Remote
Manuel Alejandro Vázquez Meza
79739406f1
Initial co...
3 hours ago
1 Commits
All
Data
Models
Notebooks
DVC
Git
.gitignore
79739406f1
Initial commit
3 hours ago
Storage Buckets
DagsHub Storage
DAGsHub
View Dataset
Connect external storage bucket (AWS, GCS, Azure, S3 Compatible)
About
Seoul bike sharing demand predictions based on Meteorological features
Collaborators 4



6.4 Documentación y comparación de configuraciones, parámetros y resultados de cada ejecución.

Validación temporal con preprocesamiento (OneHot + Imputación)

Modelo: RandomForestRegressor (n_estimators=300, random_state=42)

Preprocesamiento:

- Numéricas → imputación por mediana
- Categóricas → imputación por moda + OneHotEncoder
- Implementado en un Pipeline para evitar fuga temporal.

Validación: TimeSeriesSplit(n_splits=5)
Propósito: Estimar desempeño base sin usar información futura.
Salida: Métricas R^2 y RMSE con validación temporal y test 80/20.

Reentrenamiento con dataset enriquecido

Modelo: RandomForestRegressor (n_estimators=300, max_depth=15)
Preprocesamiento:

- OrdinalEncoder en variables categóricas.
- Sin imputación ni escalado explícito.

Validación: TimeSeriesSplit(n_splits=5)
Propósito: Entrenar con dataset ampliado (features adicionales).
Salida: R^2 y RMSE promedio, gráfico de residuales e importancia de variables.

Entrenamiento con TimeSeriesSplit usando lags

Modelo: HistGradientBoostingRegressor (learning_rate=0.08, max_depth=10, max_iter=400)

Preprocesamiento:
- Numéricas → mediana
- Categóricas → moda + OneHotEncoder
- Todo en Pipeline con ColumnTransformer

Validación: TimeSeriesSplit(n_splits=5) + test 80/20
Propósito: Incluir lags e ingeniería temporal, sin usar la fecha cruda.

GridSearch para optimización del modelo (HGBR)

Modelo: HistGradientBoostingRegressor optimizado con GridSearchCV

Hiperparámetros buscados:
- learning_rate, max_depth, max_iter, l2_regularization

Validación: TimeSeriesSplit(n_splits=5), métrica R^2
Preprocesamiento: Igual que experimento anterior.
Propósito: Afinar hiperparámetros del HGBR.

GridSearch fino + Features nuevas + Reporte visual

Modelo: HistGradientBoostingRegressor con hiperparámetros refinados.

Nuevas variables creadas:
- temp_roll_6h, rain_roll_3h, lag_24h, temp_x_solar.
- Basadas en rolling, lags e interacción temperatura × radiación.

Preprocesamiento: imputación (mediana/moda) + OneHotEncoder.

Métricas: R^2, RMSE, MAE (test 80/20).

Extras: Gráfico de residuales y ranking de importancia de variables (permutación).

Baseline ML (Random Forest)

Modelos:
- RandomForestRegressor o RandomForestClassifier (automático según el target)
- n_estimators=500, max_depth=20, min_samples_leaf=2, max_features='sqrt'

Preprocesamiento:
- Numéricas → mediana
- Categóricas → moda + OneHotEncoder
- Ordinales → OrdinalEncoder (e.g., Functioning Day)

Ingeniería adicional:
- Variables cíclicas (Hour_sin, Hour_cos)
- Seasons como nominal (por defecto)

Split: Train/Val/Test con estratificación (regresión → bins).
Métricas: MAE, MSE, RMSE, R^2 o Accuracy (según tipo de tarea).
Extras: Importancia por permutación.

XGBoost + features temporales/derivadas + LOG1P + MAPE

Modelo principal: XGBRegressor(n_estimators=700, max_depth=6, learning_rate=0.03, subsample=0.8, colsample_bytree=0.8)
Fallback: HistGradientBoostingRegressor
Preprocesamiento: igual a experimentos previos.

Ingeniería de variables:
- Derivadas: is_rush_hour, comfort_index, wind_discomfort, is_holiday_or_weekend
- Temporales: Month_sin/cos, Weekday_sin/cos, is_weekend

Target: log1p(y) → mejora estabilidad; métricas en escala original.
Métricas: MAE, MSE, RMSE, R^2, MAPE.
Extras: Importancia por permutación.

6.5 Métricas relevantes

Durante los experimentos se aplicaron métricas de evaluación adecuadas al tipo de tarea (regresión), priorizando la capacidad predictiva y la estabilidad del modelo en datos temporales.

- **R^2 (Coeficiente de determinación):** mide el grado de explicación de la variabilidad del target por parte del modelo. Se usó tanto en validación cruzada temporal (TimeSeriesSplit) como en el conjunto de test final.

- **RMSE (Root Mean Squared Error**): error cuadrático medio en escala original, sensible a outliers.

- **MAE (Mean Absolute Error):** error medio absoluto, útil para interpretar la magnitud promedio del error.

- **MAPE (Mean Absolute Percentage Error):** porcentaje de error medio relativo, implementado en el experimento con XGBoost para evaluar estabilidad relativa en demanda.

- **KS-test (Kolmogórov–Smirnov):** aplicado en la comparación de distribuciones (org_clean vs. mod_clean) para verificar que los procesos de limpieza y feature engineering no alteraran significativamente las distribuciones originales de las variables.

Las métricas se calcularon tanto en validación como en test (split temporal 80/20), asegurando que los resultados reflejen la generalización del modelo a periodos futuros.

6.6 Control de versiones de los experimentos

Se implementó un esquema de control y trazabilidad de los experimentos para garantizar reproducibilidad y seguimiento de mejoras entre versiones:

Versionado de datasets

Seguimiento de Datos: Cuando añadimos o actualizamos un dataset, como data/processed/seoul_bike_sharing_featured.csv, usamos DVC para que lo rastree.

```
# Ejemplo de cómo añadir un archivo a DVC
dvc add data/processed/seoul_bike_sharing_featured.csv
```

Esto crea un archivo data/processed/seoul_bike_sharing_featured.csv.dvc. Este pequeño archivo .dvc es el que se versiona en Git.

Almacenamiento Remoto: El archivo de datos real se sube a un almacenamiento remoto configurado para DVC con Google Drive. Esto mantiene nuestro repositorio de Git ligero y rápido.

Un commit en Git captura el estado del código y la versión de los datos (a través de los archivos .dvc).

Si un compañero de equipo quiere reproducir un experimento, solo necesita hacer:

1. Obtener el código de una versión específica
git checkout <commit_hash>

2. Sincronizar los datos correspondientes a esa versión
dvc pull

Con dvc pull, DVC lee los archivos .dvc de esa versión del código y descarga los datos exactos que se usaron en ese momento desde el almacenamiento remoto.

Flujo de nuestro proyecto

Nuestro flujo de trabajo aprovecha DVC en cada etapa:

mlops/dataset.py: Este script toma los datos crudos de data/raw y genera una versión limpia en data/interim. DVC rastrea tanto la entrada como la salida.

mlops/features.py: Toma los datos limpios de data/interim y crea el dataset final con ingeniería de características en data/processed. DVC también gestiona esta dependencia.

mlops/modeling/train.py: Utiliza el dataset de data/processed para entrenar un modelo y lo guarda en models/. El modelo resultante, como best_model.pkl, también es versionado por DVC.

Gracias a esta estructura, podemos garantizar que cada experimento registrado en MLflow está asociado a una versión específica de código, datos y modelo, haciendo que nuestros resultados sean 100% reproducibles.

Versionado de Experimentos con YAML y MLflow

Para asegurar que nuestros esfuerzos de modelado sean organizados, reproducibles y colaborativos, hemos implementado un sistema de versionado de experimentos. Este sistema se basa en la combinación de un archivo de configuración central (experiments.yaml), un script de orquestación (run_experiments.py) y el seguimiento de MLflow integrado con DagsHub.

Nuestro flujo de trabajo

El sistema se compone de tres partes principales que trabajan en conjunto:

1. experiments.yaml: Panel de control

Este archivo es el corazón de nuestro sistema de experimentación. En lugar de cambiar parámetros directamente en el código, los definimos aquí. Cada entrada de primer nivel en el YAML representa un experimento único.

```
# experiments.yaml

# Experimento base con el modelo por defecto (usará los parámetros de config.py)
baseline_hgb:
  model_name: "hist_gradient_boosting_regressor"
# Experimento que sobreescribe parámetros por defecto para probar algo específico
hgb_with_l2:
  model_name: "hist_gradient_boosting_regressor"
```

```
# Sobreescribimos la grilla de parámetros y la métrica de evaluación
  param_grid:
    model__l2_regularization: [0.0, 0.1, 0.5]
  metric: "neg_mean_squared_error"
```

- model_name: Especifica qué modelo del MODEL_REGISTRY en train.py se usará.
- param_grid, metric, etc.: Permiten sobreescribir cualquier parámetro por defecto definido en config.py. Si no se especifica, se usa el valor predeterminado.

2. run_experiments.py: Orquestador

Este script automatiza la ejecución de todos los experimentos definidos en experiments.yaml. Su lógica es simple pero poderosa:

1. Lee y parsea el archivo experiments.yaml.
2. Itera sobre cada experimento definido (ej: baseline_hgb, hgb_with_l2).
3. Para cada uno, construye y ejecuta un comando que llama a mlops/modeling/train.py, pasando los parámetros del YAML como argumentos de línea de comandos.

Para ejecutar todos los experimentos, simplemente corremos: *python run_experiments.py*

3. MLflow y DagsHub: Registro de resultados

Cada vez que train.py es ejecutado por el orquestador, se crea una nueva "corrida" (run) en MLflow. Gracias a la integración con DagsHub, esta corrida queda automáticamente asociada al commit de Git que contiene la versión del código y del experiments.yaml que se utilizó.

Dentro de la UI de MLflow en DagsHub, podemos:

- Comparar métricas como r2 o rmse entre diferentes experimentos.
- Revisar los hiperparámetros exactos que produjeron el mejor resultado.
- Acceder a los artefactos guardados, como el modelo serializado (best_model.pkl) o gráficos de evaluación.

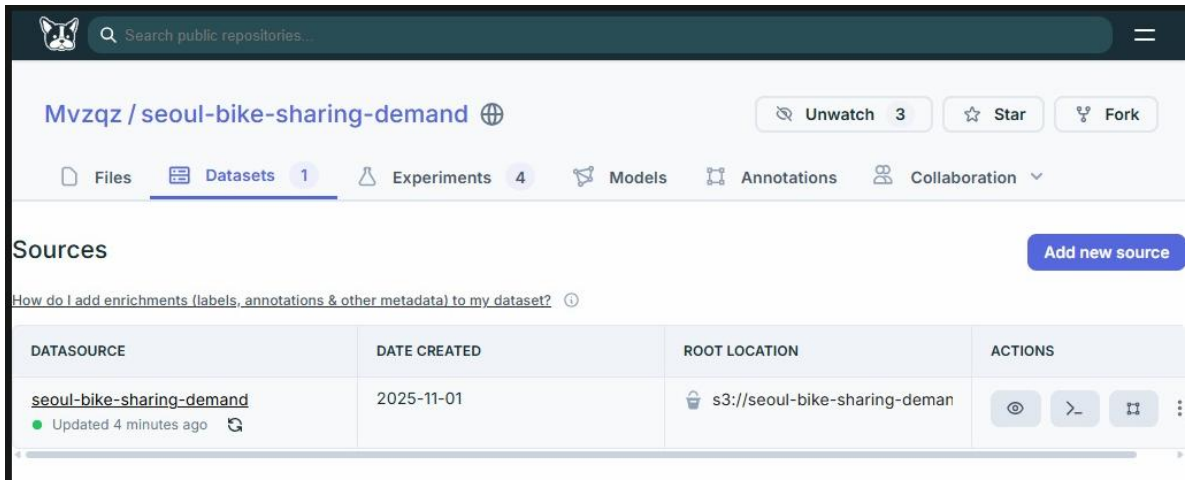
Ciclo completo

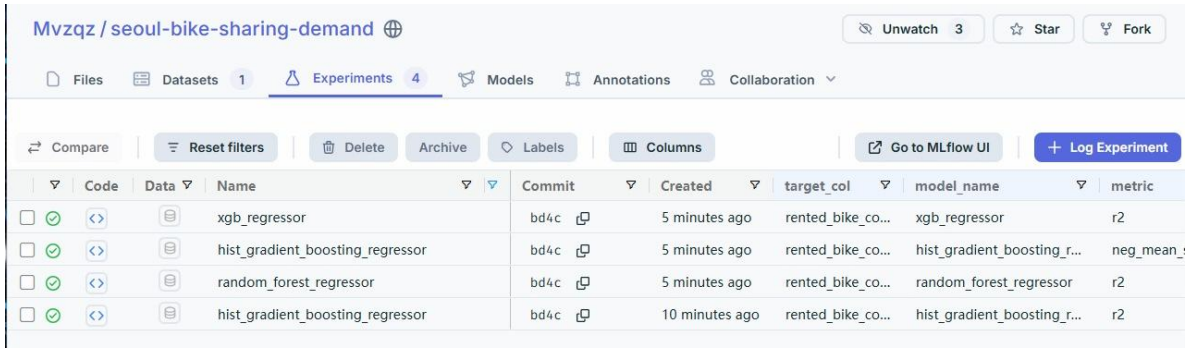
Nuestro ciclo de vida para la experimentación es el siguiente:

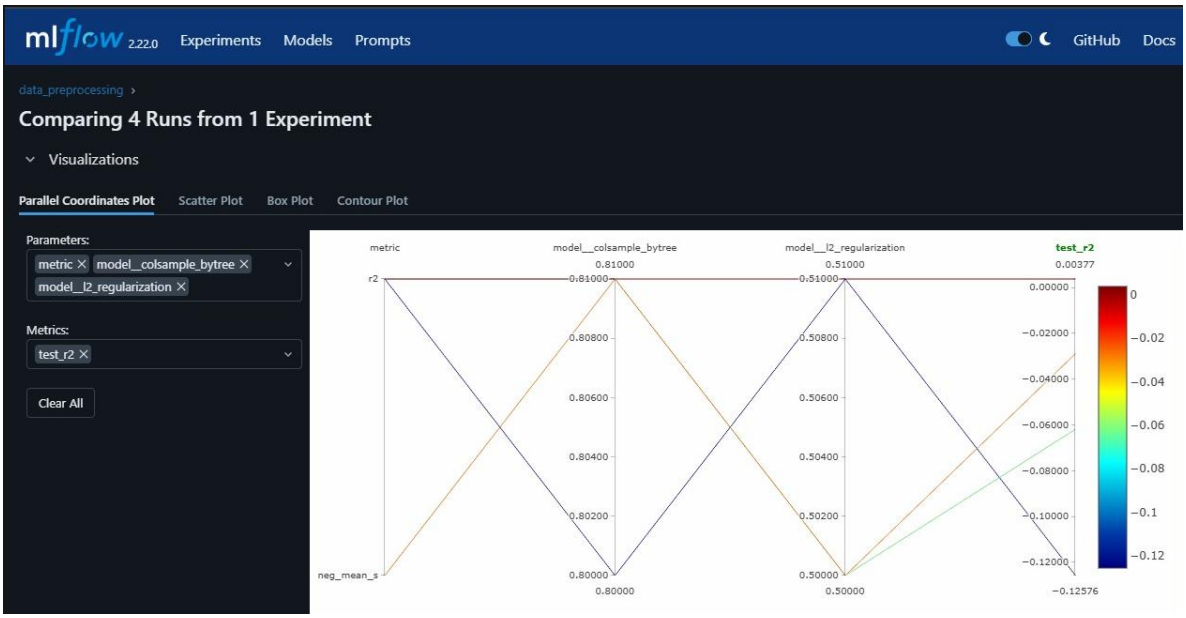
1. Definir: El ingeniero de ML añade o modifica un experimento en experiments.yaml.
2. Versionar: Se hace un commit de los cambios en Git con un mensaje descriptivo (ej: git commit -m "feat: Add experiment for XGBoost with low learning rate").
3. Ejecutar: Se corre python run_experiments.py en un entorno local o en un servidor de CI/CD.
4. Analizar: Se revisan los resultados en la pestaña "Experiments" de DagsHub para determinar los próximos pasos.

Este flujo de trabajo nos proporciona un registro histórico, versionado y auditable de todo nuestro proceso de modelado, lo cual es fundamental para un proyecto de MLOps robusto.

6.7 Registro de los modelos generados







Run details

Run ID:	269aab4114a34ce09e2eb698592405b6	1966e39a749f4b57bd8d8cedf5f4f9d1	2f93550cf3fa4429b920d1f961d58f1e	5f03a16ce35748628d088548db205e09
Run Name:	hist_gradient_boosting_regressor	xgb_regressor	random_forest_regressor	hist_gradient_boosting_regressor
Start Time:	11/01/2025, 06:14:47 PM	11/01/2025, 06:14:26 PM	11/01/2025, 06:14:01 PM	11/01/2025, 06:09:41 PM
End Time:	11/01/2025, 06:15:26 PM	11/01/2025, 06:14:42 PM	11/01/2025, 06:14:21 PM	11/01/2025, 06:13:57 PM
Duration:	38.9s	16.7s	19.7s	4.3min

Parameters

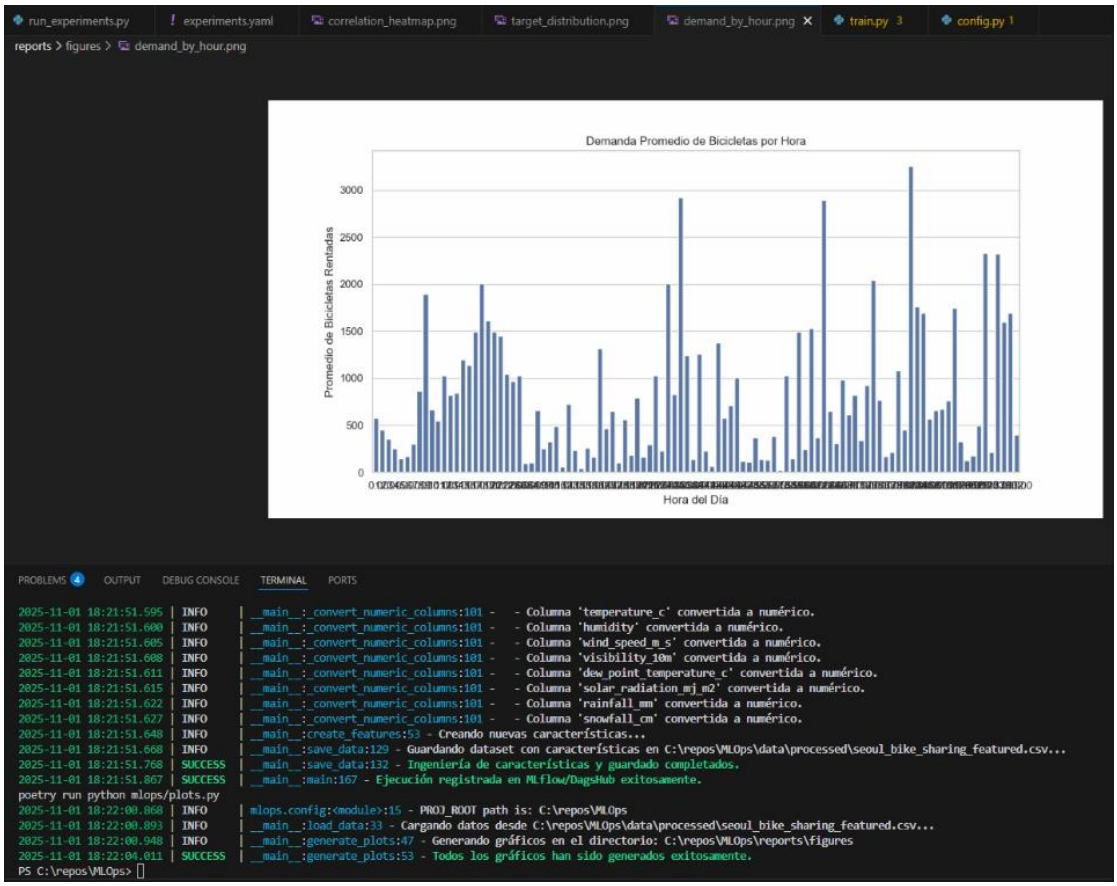
Show diff only

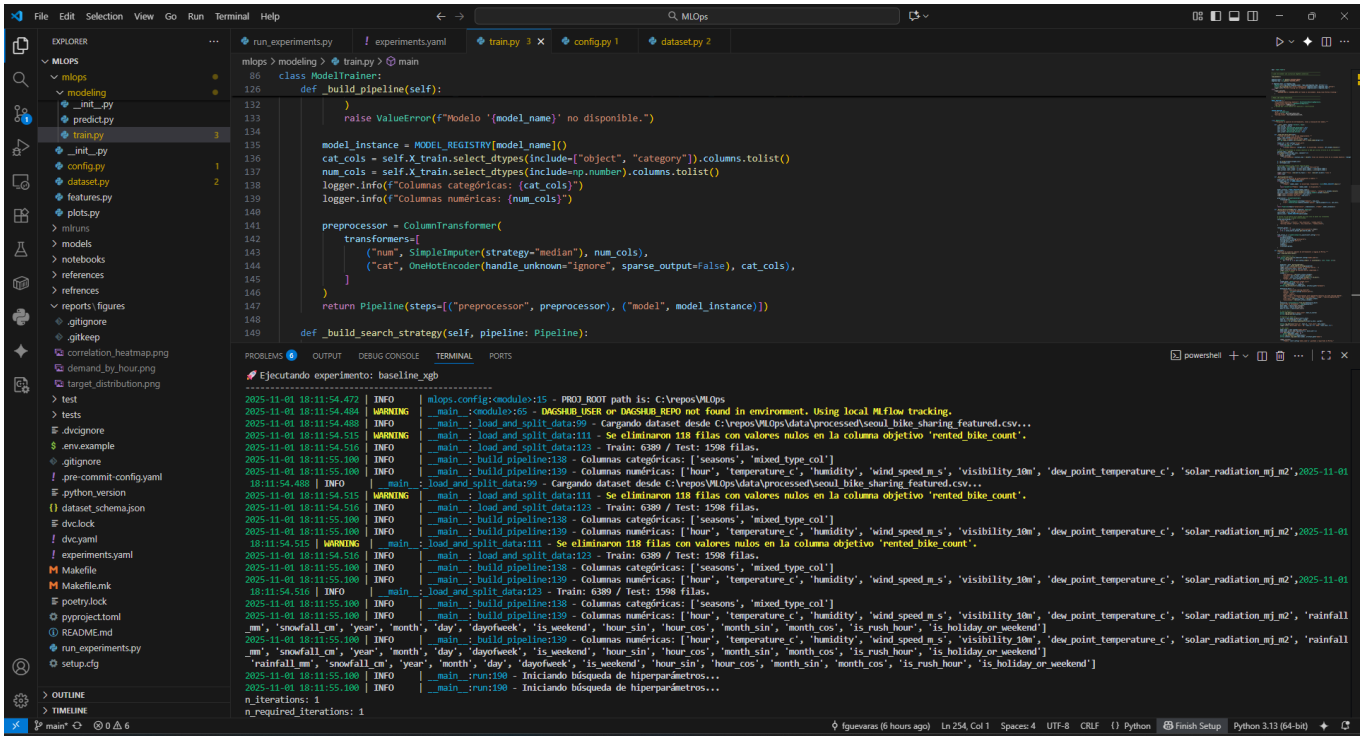
metric	neg_mean_squared_error	r2	r2	r2
model__colsample_bytree		0.8		
model__l2_regularization	0.5			0.5
model__learning_rate		0.03		0.08
model__max_depth		6	20	12
model__max_features			sqrt	
model__max_iter				500
model__min_samples_leaf			2	
model__n_estimators		700	500	
model__n_jobs		-1	-1	
model__random_state		42	42	
model__reg_lambda		1.0		
model__subsample		0.8		
model__tree_method		hist		
model_name	hist_gradient_boosting_regressor	xgb_regressor	random_forest_regressor	hist_gradient_boosting_regressor

Metrics

Show diff only

cv_best_score	-16121920.7	-0.918	0.007	-0.456
test_r2	-0.029	-0.126	0.004	-0.062
test_rmse	5354.4	5601	5269	5439.7


run_experiments.py
experiments.yaml
correlation_heatmap.png
target_distribution.png
demand_by_hour.png
train.py 3
config.py 1
reports > figures > demand_by_hour.png
Demanda Promedio de Bicicletas por Hora
Promedio de Bicicletas Rentadas
Hora del Día
PROBLEMS 4
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
2025-11-01 18:21:51.595 | INFO | __main__:convert_numeric_columns:101 - - Columna 'temperature_c' convertida a numérico.
2025-11-01 18:21:51.600 | INFO | __main__:convert_numeric_columns:101 - - Columna 'humidity' convertida a numérico.
2025-11-01 18:21:51.605 | INFO | __main__:convert_numeric_columns:101 - - Columna 'wind_speed_m_s' convertida a numérico.
2025-11-01 18:21:51.608 | INFO | __main__:convert_numeric_columns:101 - - Columna 'visibility_10m' convertida a numérico.
2025-11-01 18:21:51.611 | INFO | __main__:convert_numeric_columns:101 - - Columna 'dew_point_temperature_c' convertida a numérico.
2025-11-01 18:21:51.615 | INFO | __main__:convert_numeric_columns:101 - - Columna 'solar_radiation_mj_m2' convertida a numérico.
2025-11-01 18:21:51.622 | INFO | __main__:convert_numeric_columns:101 - - Columna 'rainfall_mm' convertida a numérico.
2025-11-01 18:21:51.627 | INFO | __main__:convert_numeric_columns:101 - - Columna 'snowfall_cm' convertida a numérico.
2025-11-01 18:21:51.648 | INFO | __main__:create_features:53 - Creando nuevas características...
2025-11-01 18:21:51.668 | INFO | __main__:save_data:129 - Guardando dataset con características en C:\repos\MLOps\data\processed\seoul_bike_sharing_featured.csv...
2025-11-01 18:21:51.768 | SUCCESS | __main__:save_data:132 - Ingeniería de características y guardado completados.
2025-11-01 18:21:51.867 | SUCCESS | __main__:main:167 - Ejecución registrada en MLflow/DagsHub exitosamente.
poetry run python mlops/plots.py
2025-11-01 18:22:00.868 | INFO | mlops.config:<module>:15 - PROJ_ROOT path is: C:\repos\MLOps
2025-11-01 18:22:00.893 | INFO | __main__:load_data:33 - Cargando datos desde C:\repos\MLOps\data\processed\seoul_bike_sharing_featured.csv...
2025-11-01 18:22:00.948 | INFO | __main__:generate_plots:47 - Generando gráficos en el directorio: C:\repos\MLOps\reports\figures
2025-11-01 18:22:04.011 | SUCCESS | __main__:generate_plots:53 - Todos los gráficos han sido generados exitosamente.
PS C:\repos\MLOps>


File Edit Selection View Go Run Terminal Help
MLOps
EXPLORER
MLOPS
run_experiments.py
experiments.yaml
train.py 3
config.py 1
dataset.py 2
mlops > modeling > train.py > main
class ModelTrainer:
def _build_pipeline(self):
raise ValueError(f"Modelo '{model_name}' no disponible.")
model_instance = MODEL_REGISTRY[model_name]()
cat_cols = self.X_train.select_dtypes(include=["object", "category"]).columns.tolist()
num_cols = self.X_train.select_dtypes(include=np.number).columns.tolist()
logger.info(f"Columnas categóricas: {cat_cols}")
logger.info(f"Columnas numéricas: {num_cols}")
preprocessor = ColumnTransformer(
transformers=[
("num", SimpleImputer(strategy="median"), num_cols),
("cat", OneHotEncoder(handle_unknown="ignore", sparse_output=False), cat_cols),
]
)
return Pipeline(steps=[("preprocessor", preprocessor), ("model", model_instance)])
def _build_search_strategy(self, pipeline: Pipeline):
PROBLEMS 6
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
Ejecutando experimento: baseline_xgb
2025-11-01 18:11:54.472 | INFO | mlops.config:<module>:15 - PROJ_ROOT path is: C:\repos\MLOps
2025-11-01 18:11:54.484 | WARNING | __main__:<module>:65 - DAGSHUB_USER or DAGSHUB_REPO not found in environment. Using local MLflow tracking.
2025-11-01 18:11:54.488 | INFO | __main__:load_and_split_data:99 - Cargando dataset desde C:\repos\MLOps\data\processed\seoul_bike_sharing_featured.csv...
2025-11-01 18:11:54.515 | WARNING | __main__:load_and_split_data:111 - Se eliminaron 118 filas con valores nulos en la columna objetivo 'rented_bike_count'.
2025-11-01 18:11:54.516 | INFO | __main__:load_and_split_data:123 - Train: 6389 / Test: 1598 filas.
2025-11-01 18:11:55.100 | INFO | __main__:build_pipeline:138 - Columnas categóricas: ['seasons', 'mixed_type_col']
2025-11-01 18:11:55.100 | INFO | __main__:build_pipeline:139 - Columnas numéricas: ['hour', 'temperature_c', 'humidity', 'wind_speed_m_s', 'visibility_10m', 'dew_point_temperature_c', 'solar_radiation_mj_m2', 'rainfall_mm', 'snowfall_cm', 'year', 'month', 'day', 'dayofweek', 'is_weekend', 'hour_sin', 'hour_cos', 'month_sin', 'month_cos', 'is_rush_hour', 'is_holiday_or_weekend']
2025-11-01 18:11:55.100 | INFO | __main__:run:190 - Iniciando búsqueda de hiperparámetros...
2025-11-01 18:11:55.100 | INFO | __main__:run:190 - Iniciando búsqueda de hiperparámetros...
n_iterations: 1
n_required_iterations: 1

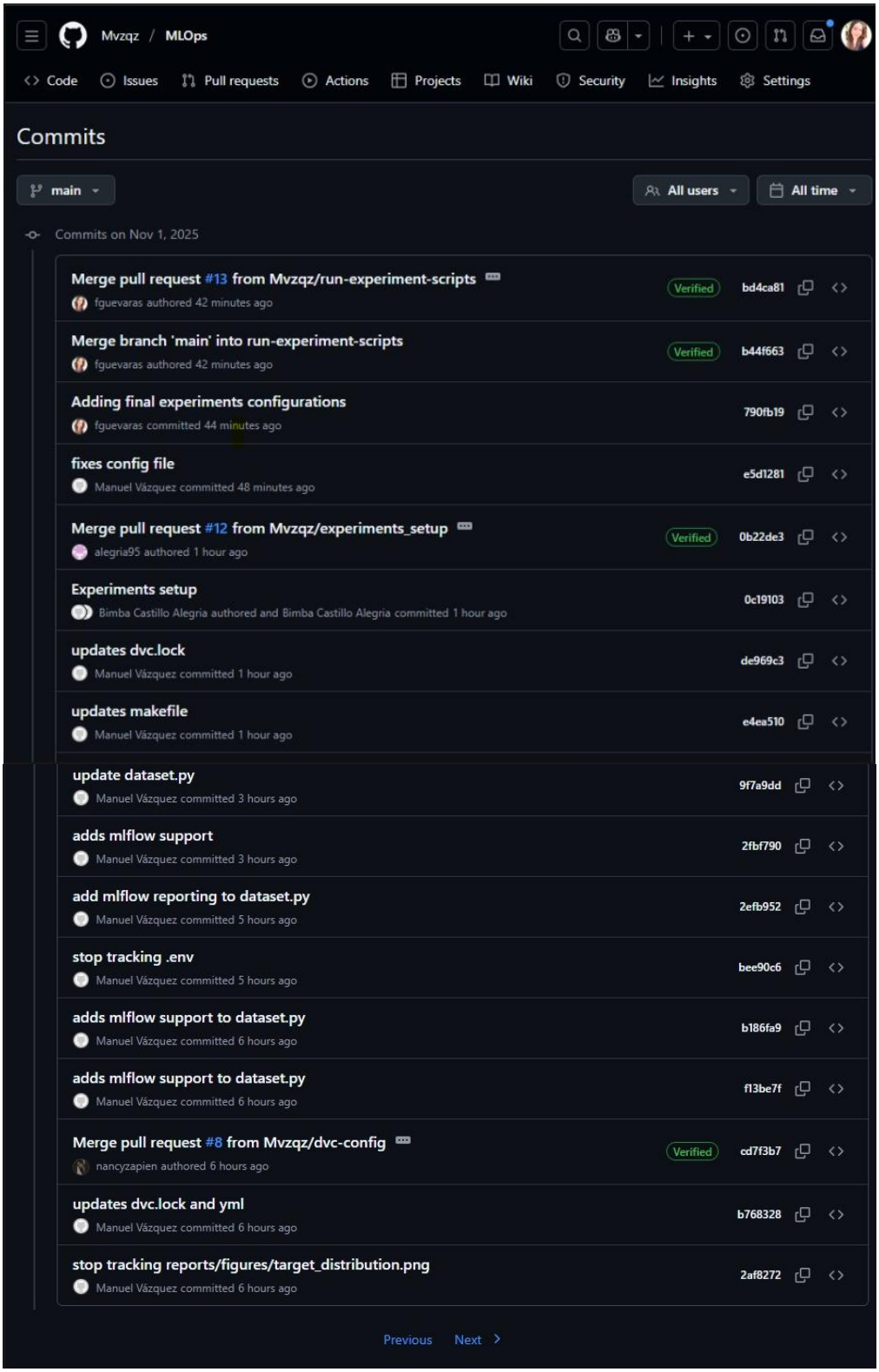
7. INTERACCIONES ENTRE ROLES

Etapa	Tareas por etapa	Tareas por rol	Entregables
Estructuración del proyecto con Cookiecutter.	**Tarea:** Implementar una estructura de proyecto estandarizada. **Instrucciones:** - Descargar y utilizar la plantilla de Cookiecutter para proyectos de ML. - Implementar el esquema de directorios y archivos propuesto en tu propio proyecto siguiendo la plantilla de Cookiecutter - Mantener una organización clara y consistente, de acuerdo con el template.	**Software Engineer:** Liderar la descarga, aplicación y configuración inicial de la plantilla. Definir los *scripts* de *setup* del nuevo entorno. **Data Engineer:** Migrar los archivos de datos versionados (.dvc files) y los *scripts* de carga/limpieza. **ML Engineer:** Migrar el código del modelo y la configuración de versiones.	Repositorio: Nueva estructura de directorios y archivos siguiendo el template Cookiecutter. Configuración: Archivos de configuración de proyecto estandarizados (setup.py, Makefile, etc). Documento: Actualización de la documentación del proyecto en el repositorio (README.md).
Estructuración y Refactorización del Código	Tarea: Mejorar la organización y mantenibilidad del código. Instrucciones: - Organiza el código en módulos y funciones con responsabilidades bien definidas. - Aplica principios de programación orientada a objetos (POO) cuando sea pertinente. - Refactoriza el código existente para mejorar su eficiencia,	**Software Engineer:** Definir los estándares de refactorización y la arquitectura de módulos (src/). Revisar la calidad del código. **Data Engineer:** Refactorizar el código de carga y *Feature Engineering* en funciones/clases reutilizables dentro de src/data o src/features. **Data Scientist:** Validar que la lógica de *Feature Engineering* se mantenga consistente después de la refactorización. **ML Engineer:**	Código: Módulos de Python organizados en el directorio src/ (src/data.py, src/features.py, src/model.py). Documento: Bitácora de Refactorización clave (demostrando modularidad y principios de POO).

Etapa	Tareas por etapa	Tareas por rol	Entregables
	legibilidad, escalabilidad y mantenimiento a largo plazo.	Refactorizar el código de entrenamiento y evaluación en un módulo de modelado dentro de src/models.	
Aplicación de Mejores Prácticas de Codificación en el Pipeline de Modelado	**Tarea:** Incorporar buenas prácticas en las etapas del pipeline usando SciKit-Learn. **Instrucciones:** - Implementar un pipeline de Scikit-Learn que automatice las etapas de preprocesamiento, entrenamiento y evaluación. - Documentar cada paso, asegurando que sea claro, reproducible y entendible por terceros.	**ML Engineer:** Diseñar y construir el Pipeline de Scikit-Learn (usando Pipeline y ColumnTransformer). Asegurar que los pasos sean reproducibles. **Data Scientist:** Adaptar las transformaciones definidas en la Fase 1 a los transformadores personalizados de Scikit-Learn (clases que implementan fit/transform). **Software Engineer:** Revisar que el código del Pipeline cumpla con los estándares de codificación.	Código: *Script* de modelado principal (train_pipeline.py) que utiliza la clase Pipeline de Scikit-Learn. Configuración: Archivos de configuración de parámetros actualizados para el nuevo *script*.
Seguimiento de Experimentos, Visualización de Resultados y Gestión de Modelos	**Tarea:** Registrar y versionar experimentos, visualizar resultados y gestionar modelos de forma ordenada. **Instrucciones:** - Utilizar herramientas como MLflow, DVC para dar seguimiento a los experimentos. - Documentar y comparar configuraciones, parámetros y	**ML Engineer:** Configurar el servidor de MLflow Tracking o el repositorio de DVC. Diseñar la estructura de los experimentos. Ejecutar el *Hyperparameter Tuning* y registra cada *run*. **Data Scientist:** Definir los rangos de hiperparámetros a experimentar y las métricas a monitorear. **Data Engineer:** Asegurar que los datos versionados por DVC se enlacen	Configuración: Archivos de configuración de MLflow (mlruns/ directory y configuración de servidor). Artefactos: Visualización de MLflow UI (comparación de *runs* con métricas y parámetros). Registro de Modelos: El mejor modelo fue registrado y etiquetado en el MLflow

Etapa	Tareas por etapa	Tareas por rol	Entregables
	resultados de cada ejecución. - Registrar métricas relevantes y asegurar el control de versiones de los experimentos. Utilizar las herramientas de visualización de MLFlow para presentar los resultados de manera clara y comprensible. Mantener un registro actualizado de los modelos generados, incluyendo: o Versión o Hiperparámetros o Métricas de evaluación o Resultados relevantes	correctamente con los *runs* de MLflow (artefactos). **SRE:** Documentar la infraestructura de persistencia (servidor MLflow o *backend store*).	Model Registry y rastreado con DVC.

7.1 Evidencias de participación



8. CONCLUSIONES Y REFLEXIÓN FINAL

Esta segunda fase ha sido un paso transformador para el proyecto, elevándolo de un prototipo exploratorio a un pipeline de Machine Learning estructurado, reproducible y alineado con las prácticas profesionales de MLOps. A continuación, se resumen los logros, las áreas de mejora y los próximos pasos a seguir.

8.1 Logros clave de la fase 2

Profesionalización de la estructura del proyecto
Se abandonó el enfoque de notebook monolítico y se adoptó una estructura de directorios estandarizada mediante cookiecutter-data-science. Esto proporciona una separación clara entre datos, código fuente, modelos y reportes, mejorando drásticamente la organización y facilitando la colaboración y escalabilidad futura.

Refactorización y modularidad del código
El código fue refactorizado del notebook a módulos de Python (.py) con responsabilidades únicas (limpieza, ingeniería de características, entrenamiento). La aplicación de principios de Programación Orientada a Objetos (POO) con clases como DatasetProcessor y ModelTrainer ha encapsulado la lógica, haciendo el código más mantenible, reutilizable y fácil de probar.

Implementación de un Pipeline de modelado robusto
Se construyó un scikit-learn Pipeline que automatiza la secuencia de preprocesamiento y entrenamiento. El uso de ColumnTransformer y su integración con GridSearchCV y TimeSeriesSplit garantiza que no haya fuga de datos (data leakage), lo que resulta en métricas de evaluación más fiables y representativas del rendimiento del modelo en datos no vistos.

Seguimiento sistemático de experimentos
Se integró MLflow en el script de entrenamiento para registrar automáticamente cada experimento. Ahora, cada ejecución almacena sus hiperparámetros, métricas de validación y prueba, y el propio modelo como un artefacto. Esto no solo proporciona una trazabilidad completa, sino que también permite comparar visualmente los resultados a través de la UI de MLflow para tomar decisiones informadas sobre qué modelo es el mejor.

8.2 Mejoras para la siguiente fase

Las siguientes áreas son oportunidades clave de mejora:

Automatización (CI/CD): Implementar un pipeline de Integración Continua y Despliegue Continuo (CI/CD) con herramientas como GitHub Actions. Esto permitiría automatizar las pruebas, el reentrenamiento del modelo y, potencialmente, el despliegue, cada vez que se realice un cambio en el código o los datos.

Monitoreo en Producción: Una vez que un modelo se despliega, es crucial monitorear su rendimiento. Se podría implementar un sistema para detectar el "model drift" (cuando el

rendimiento del modelo decae con el tiempo) y el "data drift" (cuando los datos de entrada cambian significativamente).

Versionado de Datos: Aunque se usó DVC para el versionado inicial, se puede formalizar su uso para versionar no solo los datos crudos, sino también los datasets procesados y los modelos, creando un linaje de datos completo y auditable.

Pruebas Unitarias y de Integración: Añadir un conjunto de pruebas formales (pytest) para cada módulo de código (dataset.py, features.py, etc.) para garantizar que los cambios futuros no rompan la funcionalidad existente.

8.3 Próximos Pasos

El siguiente paso lógico en el ciclo de MLOps es el despliegue del modelo para que pueda ser consumido por otras aplicaciones. El plan inmediato es:

Empaquetar el modelo seleccionando el mejor modelo del registro de MLflow y empaquetarlo como un artefacto listo para ser servido.

Crear una API REST desarrollando una API simple utilizando un framework como FastAPI. Esta API cargará el modelo y expondrá un endpoint (ej. /predict) que reciba los datos de entrada y devuelva la predicción de la demanda de bicicletas.

Contenerización creando un Dockerfile para empaquetar la aplicación de la API y todas sus dependencias en una imagen de contenedor, garantizando un entorno de ejecución consistente.

Despliegue realizando un despliegue inicial del contenedor, ya sea localmente con Docker o en una plataforma en la nube (como Heroku, AWS, o Google Cloud) para hacer que el modelo sea accesible a través de internet.

9. ANEXOS

9.1 Liga presentación

Fase 2 Equipo 44 - Presentación

9.2 Liga al video

Fase 2- Equipo 44 - Video

9.3 Liga al repositorio

Fase 2- Equipo 44 - Repositorio

9.4 DVC

Fase 2 - Equipo 44 - DVC

9.5 ML Flow

Fase 2 - Equipo 44 - MLFlow

10. REFERENCIAS

Cookiecutter Data Science. (s. f.). *Using the template.* DrivenData. https://cookiecutter-data-science.drivendata.org/using-the-template/

Chen, T., & Guestrin, C. (2016). *XGBoost: A scalable tree boosting system.* In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (pp. 785–794). https://doi.org/10.1145/2939672.2939785

Data Version Control (DVC). (2024). *DVC documentation.* Iterative.ai. https://dvc.org/doc
Hunter, J. D. (2007). *Matplotlib: A 2D graphics environment. Computing in Science & Engineering, 9*(3), 90–95. https://doi.org/10.1109/MCSE.2007.55

Kreuzberger, D., Kühl, N., & Hirschl, S. (2023). *Machine learning operations (MLOps): Overview, definition, and architecture. IEEE Access, 11,* 31866–31880. https://doi.org/10.1109/ACCESS.2023.3242724
McKinney, W. (2010). *Data structures for statistical computing in Python.* In *Proceedings of the 9th Python in Science Conference* (pp. 51–56). https://doi.org/10.25080/Majora-92bf1922-00a

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... & Duchesnay, É. (2011). *Scikit-learn: Machine learning in Python. Journal of Machine Learning Research, 12,* 2825–2830. https://jmlr.org/papers/v12/pedregosa11a.html

UCI Machine Learning Repository. (2020). *Seoul bike sharing demand data set.* University of California, Irvine. https://archive.ics.uci.edu/dataset/560/seoul+bike+sharing+demand

Waskom, M. (2021). *Seaborn: Statistical data visualization. Journal of Open Source Software, 6*(60), 3021. https://doi.org/10.21105/joss.03021