Proyecto Inteligencia Artificial y Big Data

Grupo B

Héctor Fernández Pimienta, Ignacio López García y Francisco José Puertas Teba

FASE 3: PRODUCTO DE DATOS, BIG DATA & MLOPS

Introducción y Objetivos del Proyecto

Propósito y Objetivos del Proyecto

El propósito central de nuestro proyecto es transformar un prototipo de clustering de datos musicales en un Producto de Datos (Data Product) robusto, automatizado y escalable. Hemos trabajado sobre un dataset de Spotify con el objetivo de segmentar las canciones basándonos en sus características acústicas (tempo, danceability, valence, etc.), permitiendo así la creación de perfiles de usuario o listas de reproducción optimizadas.

Nuestros objetivos para esta Fase 3 se centran en la implementación de la metodología MLOps para asegurar:

- **Robustez y Servicio**: Diseñar e implementar un endpoint de inferencia de baja latencia mediante una API con contrato estricto.
- **Automatización**: Establecer una Orquestación (DAG) para que el retraining del modelo y la ingesta de datos sean procesos automáticos y reproducibles.
- **Calidad y Control:** Implementar un sistema de Monitorización para las métricas de negocio y de modelo.
- **Cumplimiento:** Definir la estrategia de Pruebas, Seguridad y Costes para un despliegue viable.

De Prototipo a Producto Robusto

Hemos pasado de tener dos Scripts aislados (Pipeline.py y Entrenamiento.py) a concebir un sistema completo que integra:

- **Pipeline de Datos y ML:** Implementación con PySpark y scikit-learn.
- **Modelo Persistente:** Serializado en kmeans_spotify_model.pkl.
- **API de Inferencia (Diseño):** Basada en FastAPI para servir predicciones en tiempo real.
- **Orquestación (Diseño):** Flujo de trabajo reproducible con Apache Airflow.

- **Control de Calidad:** Pruebas unitarias con Pytest (test_pipeline.py, test_entrenamiento.py).

Stack Tecnológico.

Componente	Herramienta	Archivos Implementados	Justificación
Big Data / ETL	**PySpark**	*Pipeline.py*	Procesamiento escalable de la ingesta, limpieza de nulos y aplicación de reglas de negocio (outliers).
Machine Learning	scikit-learn, K-Means	*Entrenamiento.py, *.pkl*	Algoritmo adecuado para la segmentación no supervisada; *joblib* para la persistencia del modelo.
Persistencia	Parquet, *joblib*	*datasets/*.parquet*	Formato eficiente para la lectura y escritura en PySpark y Pandas.
Testing	Pytest	*requirements.txt, test_*.py*	Garantiza la calidad del código, especialmente la estandarización y la división de datos.
MLOps	FastAPI, Airflow, Docker	*(Diseño)*	Frameworks estándar para el servicio de inferencia, la automatización del pipeline y el despliegue.

Arquitectura de la Solución

Diseño de la Arquitectura MLOps

La solución MLOps que hemos diseñado se basa en un ciclo de vida continuo que se garantiza la frescura del modelo y su calidad en producción. El flujo se divide en tres fases principales: Data/Training, Serving e Integración/Control.

Diagrama de la Arquitectura Propuesta

- **Fuente de Datos:** Spotify Dataset (simulado con CSV).
- **Fase 1:** Data & Training (Orquestada por Airflow):
  - Airflow DAG (Scheduler) lanza el pipeline.
  - **Limpieza/División:** *Pipeline.py* (PySpark) ingiere, limpia, aplica filtros de *outliers* y guarda *X_train.parquet.*
  - **Entrenamiento/Evaluación:** *Entrenamiento.py* (Pandas/scikit-learn) lee Parquet, estandariza, entrena K-Means y evalúa (Silhouette Score).
  - **Model Registry/Storage:** El modelo (kmeans_spotify_model.pkl) se registra y almacena.
- **Fase 2:** Serving (FastAPI/Docker):
  - La imagen Docker contiene la API que carga el *.pkl* en memoria.
  - **Inferencia:** El *endpoint /predict* recibe una petición JSON y devuelve el ID del *cluster*.
- **Fase 3:** Monitorización & Feedback:
  - **Servicio:** Latencia y Tasa de Error (Prometheus/Grafana).
  - **Modelo:** Silhouette Score y Data Drift (Dashboard BI).

Decisiones Tecnológicas Detalladas

A. PySpark para la Ingesta (Big Data)

Hemos optado por **PySpark** en *Pipeline.py* para la etapa de Ingesta y Limpieza. Esta decisión se justifica por la capacidad de Spark para manejar volúmenes de datos que superan la memoria de un único nodo (*Big Data*). Aunque el dataset actual es pequeño, la arquitectura está preparada para escalar.

B. Scikit-learn y K-Means

Para la modelización, elegimos **K-Means** por ser un algoritmo de *clustering* no supervisado que se alinea con el objetivo de segmentación de canciones. Utilizamos **scikit-learn** en *Entrenamiento.py* por su madurez y rendimiento para los cálculos de entrenamiento (una vez que los datos han sido preparados por Spark).

C. Persistencia y Serialización

- **Datos:** Los conjuntos de datos limpios (*X_train, X_test*) se guardan en formato **Parquet**. Este formato binario y columnario es altamente eficiente para PySpark y Pandas, optimizando la entrada a la fase de entrenamiento.
- **Modelo:** El modelo entrenado se serializa con *joblib (joblib.dump (kmeans, nombre_modelo))*, tal como se ve en *Entrenamiento.py*, siendo la práctica estándar en scikit-learn para un servicio rápido.

Pipeline de Datos y Entrenamiento

Ingesta y Limpieza de Datos (Función *limpieza_datos* en *Pipeline.py*)

La limpieza de datos es la primera fase crítica. La función *limpieza_datos* ejecuta los siguientes pasos, orquestados por PySpark:

1. **Inicio de la Sesión Spark:** Inicializamos la sesión (*iniciar_spark*) para habilitar el procesamiento distribuido.
2. **Lectura:** Leemos el archivo *sportify_data.csv* y lo cargamos en un DataFrame de Spark
3. .**Eliminación de Metadata:** Eliminamos la columna índice *_c0*, que no aporta valor al modelo.
4. **Manejo de Nulos:** Aplicamos una estrategia de eliminación de filas que contengan cualquier valor nulo (*df_spotify = df_spotify.na.drop()*), ya que el volumen de datos lo permite y simplifica el entrenamiento posterior.
5. **Reglas de Negocio (Filtros):** Aplicamos filtros clave para mejorar la calidad del *clustering*:
    - Se filtran los géneros musicales (aunque en el código se muestra la intención de filtrado de géneros, el ejemplo implementado se centra en la limpieza básica y límites).

- **Filtro de Outliers:** Hemos limitado el *tempo* a un rango razonable *(f.col('tempo') >= 40) & (f.col('tempo') <= 250))* para evitar *clusters* sesgados por valores extremos de tempo.

6. **Muestreo / Conversión:** Al final, convertimos el *DataFrame* de Spark a Pandas y realizamos un muestreo si es necesario (*df_spotify.limit(1000)*).

División y Preprocesamiento del Dataset.

A. División (Función *dividir_dataset* en *Pipeline.py*)

Una vez limpio, El *DataFrame* se divide para evitar el data *leakage*.

1. **Separación de Variables:** El target (*popularity*) se separa de la *features* (*X*).
2. **División:** Utilizamos *train_test_split* de scikit-learn con un *test_size=0.2* (80% entrenamiento, 20% prueba) y *random_state=42* para garantizar la reproducibilidad.
3. **Persistencia:** Los cuatro conjuntos resultantes (*X_train, X_test, y_train, y_test*) se guardan en la carpeta *datasets/* como archivos Parquet.

Preprocesamiento de Features (Función *Cargar_Datos_Ruta* en *Entrenamiento.py*)

La preparación final ocurre justo antes de entrenar:

1. **Carga:** Leemos *X_train.parquet* y *x_test.parquet*.
2. **Selección de Features:** Eliminamos las columnas de texto (*'artist_name'*, *'track_name'*, etc.) y seleccionamos únicamente las *features* de *clustering* para el entrenamiento, tal como se define en el código*: clust_feats = ['instrumentalness', 'speechiness', 'danceability', 'valence', 'tempo']*.
3. **Estandarización:** Aplicamos *StandardScaler*. Este paso es fundamental para K-Means, ya que el algoritmo se basa en la distancia euclidiana y las features deben estar en la misma escala (media cero y desviación estándar unitaria).

Entrenamioento del Modelo (Función *entrenamiento* en *Entrenamiento.py*)

La función *entrenamiento* recibe los datos estandarizados y es la encargada de generar y persistir el modelo final.

1. **Inicialización:** Se crea una instancia de *KMeans* con *n_clusters=k* y *random_state=42* para asegurar la repetibilidad.
2. **Ajuste (Fit):** Ejecutamos *kmeans.fit(x_train)*.
3. **Serialización:** El modelo entrenado se guarda inmediatamente con *joblib.dump(kmeans, nombre_modelo)* en la ruta *modelos/kmeans_spotify_model.pkl*. Esto lo convierte en un artefacto listo para el despliegue.
4. **Evaluación:** Calculamos y registramos las métricas de calidad del clustering:
    - **WCSS (Inertia):** *kmeans.inertia_* (mide la cohesión dentro de los clusters).
    - **Silhouette Score:** *silhouette_score(x_train, kmeans.labels_)* (mide la separación entre *clusters*, es un valor entre -1 y 1).

Servicio de Inferencia (API) y Contrato

Para llevar el modelo a producción y que pueda ser consumido por una aplicación externa (ej. una *playlist* generator), hemos diseñado un servicio de inferencia mediante una **API REST**.

Diseño del Servicio API

Hemos elegido **FastAPI** como *framework* por su rendimiento asíncrono (*non-blocking*) y su capacidad de generar automáticamente la documentación **OpenAPI (Swagger UI)**.

- **Estrategia de Carga:** El modelo (*kmeans_spotify_model.pkl*) debe cargarse durante el arranque del servicio. Esto evita la latencia de carga en cada petición, garantizando un tiempo de respuesta óptimo para la inferencia.
- **Preprocesamiento en la API:** La API debe replicar la lógica de preprocesamiento de *Entrenamiento.py*. Esto significa que los datos de entrada (*raw features*) deben ser estandarizados utilizando los mismos parámetros de *StandardScaler* que se ajustaron durante el entrenamiento.

Contrato de la API y Validación (Pydantic/OpenAPI)

El contrato del servicio se define mediante el estándar OpenAPI, y la validación se implementa con **Pydantic** para garantizar la integridad de los datos de entrada.

A. Modelo de Request (Input)

El modelo de Pydantic debe asegurar que se reciben las 5 features clave, y que son valores flotantes:

Campo	Tipo	Requisito de Pydantic	Descripción
Instrumentalness	Float	Mayor o igual a 0.0	Proporción de música sin voz.
Speechiness	Float	Mayor o igual a 0.0	Presencia de palabras habladas.
Danceability	Float	Entre 0.0 y 1.0	Qué tan bailable es la canción.
Valence	Float	Entre 0.0 y 1.0	Positividad musical (alegría).
Tempo	Float	Mayor de 40	El ritmo o velocidad de la canción (BPM).

B. Definición del *Endpoint.*

Ruta: */v1/predict/cluster* **Método:** *POST*

Lógica del Endpoint:

1. Recibir el cuerpo JSON.
2. Pydantic valida automáticamente el formato.
3. Aplicar el StandardScaler (cargado junto al modelo).
4. Llamar a *kmeans_model.predict(datos_escalados)*.Devolver el ID del *cluster* como un campo *cluster_id*.

C. Manejo de Errores

Implementaremos códigos HTTP para una comunicación clara con el cliente:

- **200 OK**: Predicción exitosa.
- **422 Unprocessable Entity:** Error de validación de Pydantic (si el cliente envía datos con tipo o rango incorrecto).
- **500 Internal Server Error:** Errores internos, como un fallo al cargar el modelo o un problema durante la inferencia.

Orquestación y Despliegue

Diseño del DAG de Retraining (Prefect)

Para automatizar el retraining y la actualización de los datos, hemos diseñado un DAG (Grafo Dirigido Acíclico) que se ejecutará semanalmente mediante Prefect.

Antes de nada se va ha explicar el proceso de orquestación y porque se va cambiado de Apache Airflow a Prefect. El plan original era usar Airflow pero debido a su funcionamiento con contenedores daba problemas con Spark por depender de java, para agilizar el proceso se paso a Prefect por comodidad y gracias a eso se pudo terminar la Orquestación manual, ejecutando en orden los scripts de Pipeline.py y Entrenamiento.py, se tiene en planes mejorar y hacer un Flow mas complejo como se indica en la tabla de abajo, pero de momento comentar los problemas y porque el Flow es manual, al crear un despliegue se configura como va ha tratar el proyecto, el funcionamiento de Prefect es copiar o clonar el repositorio en una carpeta temporal pero eso da problemas a la hora de la persistencia de datos porque por cada tarea crea un proyecto temporal, una solución que se encontró fue guardar ficheros como los dataframes o modelos en una carpeta externa del proeyecto como C:\ pero no se consiguió de manera que fuese fácil de exportar a otras maquinas, por eso tampoco se hizo en un contenedor Docker.

Flujo del DAG (spotify_clustering_dag):

Tarea	Archivo / Función	Descripción
T1_Ingesta_y_Limpieza	Pipeline.py:limpieza_datos	Inicia Spark, lee el CSV de la fuente, aplica la limpieza (nulos y filtros de *outliers*), y persiste los conjuntos *X_train.parquet* y *X_test.parquet*.
T2_Entrenamiento_Modelo	Entrenamiento.py:entrenamiento	Carga los Parquet, aplica la estandarización, entrena el modelo K-Means y calcula WCSS/Silhouette. Guarda *kmeans_spotify_model.pkl*.
T3_Evaluacion_Calidad	Módulo de Evaluación	Lee el *Silhouette Score* de la Tarea 2. **Comprueba si el score supera un umbral mínimo** (ej. 0.35). Si no lo supera, el proceso se detiene y lanza una alerta.
T4_Registro_Despliegue	Script de Despliegue	Si la calidad es aceptable, el nuevo *kmeans_spotify_model.pkl* se promueve al entorno de producción (ej. se copia al volumen donde la API lo consume).

T5_Notificacion_Fallo	Email/Slack Alert	Tarea ejecutada si T3 falla. Alerta al equipo de MLOps sobre la degradación de la calidad del modelo.

Gestión de Dependencia y Entorno

Hemos centralizado las dependencias en el archivo *requirements.txt*, el cual será la base para la construcción de los contenedores Docker tanto para la API de inferencia como para los *workers* de Airflow. Las dependencias críticas son: *pyspark, scikit-learn, pandas, joblib y pytest*.

La **Contenerización (Docker)** es esencial para asegurar la portabilidad y evitar el *dependency hell*. Creamos una imagen base de Python que instala todas las librerías del *requirements.txt* y añade nuestros scripts (*Pipeline.py, Entrenamiento.py*) y el modelo.

Plan de Despliegue

La solución final se desplegará mediante contenedores:

1. **API de Inferencia:** Despliegue continuo de la imagen Docker de FastAPI en una plataforma de contenedores (ej. AWS ECS/Fargate o Google Cloud Run). Esto permite el autoescalado horizontal para manejar picos de tráfico.
2. **Orquestador:** Despliegue de Apache Airflow, que mantendrá la lógica del DAG separada del servicio de inferencia, garantizando que el retraining (proceso pesado de Big Data) no afecte a la latencia de la API.

Monitorización y Dashboard BI

La monitorización es clave para el MLOps. No solo debemos monitorizar el servicio de la API, sino también la calidad del modelo de *clustering* en el tiempo.

Monitorización del Servicio (Prometheus/Grafana)

Utilizaremos **Prometheus** para la recolección de métricas de la API de FastAPI y **Grafana** para la visualización.

Métrica	KPI / Umbral	Justificación
Latencia de Inferencia	P95 < 50 ms	Asegurar una experiencia de usuario fluida. Alerta si se supera el umbral.
Tasa de Error	Error Rate (códigos 5xx) < 0.5%	Controlar fallos del servicio (errores en el modelo o en la infraestructura).
Volumen de Peticiones	Tasa de solicitudes por minuto	Monitorizar el tráfico y planificar el autoescalado de la API.

Monitorización de la Calidad del Modelo (Data/Model Drift)

Necesitamos detectar si el modelo se vuelve obsoleto (o si los datos de entrada cambian) con el tiempo.

- **Métrica de Calidad (Model Drift):** El **Silhouette Score** (calculado en *Entrenamiento.py*) se registra tras cada *retraining*. Alerta de Grafana si el *score* cae más de un 10% respecto a la versión anterior.

- **Métrica de Datos (Data Drift):** Monitorizaremos la **media y desviación estándar** de las 5 *features* clave (*tempo*, *danceability*, etc.) en la producción. Una desviación significativa respecto a los valores de entrenamiento es un indicador de que el modelo debe ser reentrenado.

Dashboard BI (Streamlit/Metabase)

Implementaremos un *Dashboard* con 3 vistas accionables para los analistas de negocio:

Vista	Objetivo de Negocio	Fuente de Datos
1. Perfil del Clúster	Interpretar el significado de cada segmento (ej. Clúster 3 = "Canciones de Baile de Alto Tempo").	Medias de *danceability*, *tempo*, etc. por *Cluster ID*.
2. Distribución de Segmentos	Entender el peso y la importancia de cada *cluster* en la base de datos de canciones actual.	Frecuencia absoluta y relativa de canciones por *Cluster ID*.
3. Evolución del Modelo	Indicador de obsolescencia.	Gráfico de línea con el *Silhouette Score* a lo largo del tiempo (cada *retraining*).

Pruebas, Seguridad y Costes

Estrategia de Pruebas

Hemos adoptado un enfoque de pruebas en tres niveles para garantizar la estabilidad del producto.

A. Pruebas Unitarias (Pytest)

Hemos implementado las pruebas unitarias en *test_pipeline.py* y *test_entrenamiento.py*:

- ***test_entrenamiento.py:*** Una prueba crucial verifica la **estandarización** de los datos. Comprobamos que, tras la ejecución de *Cargar_Datos_Ruta*, la media de *X_train_scaled* en cada columna sea próxima a cero y la desviación estándar sea próxima a uno (*assert np.allclose(medias, 0, atol=1e-6)*), confirmando que el *StandardScaler* funciona correctamente antes de K-Means.
- ***test_pipeline.py:*** Verificamos la división de datos, asegurando que el *DataFrame* original no es modificado y que los archivos Parquet se generan con el formato correcto.

B. Pruebas de Integración y Contrato

- **Integración:** Probaremos que Pipeline.py genera correctamente los archivos Parquet y que Entrenamiento.py es capaz de leerlos sin errores.
- **Contrato:** Utilizaremos la documentación de OpenAPI (generada por FastAPI) para crear pruebas que garanticen que la API siempre acepta el modelo de Request definido en Pydantic y devuelve el modelo de Response esperado.

C. Pruebas de Carga (Locust / k6)

Diseñaremos una prueba de estrés para el endpoint */v1/predict/cluster* simulando:

- **Carga:** 100 usuarios concurrentes (usuarios virtuales).
- **Duración:** 5 minutos.
- **KPI de Aceptación:** La latencia P95 debe mantenerse por debajo del umbral de servicio (50 ms).

Seguridad y Aspectos Legales

Hemos integrado controles básicos de seguridad para proteger el servicio y la información.

- **Gestión de Secretos:** Implementamos un archivo *.env.example* para la definición de secretos (ej. *MONGODB_URI*, *SECRET_KEY*). En un entorno de producción, estos se gestionan mediante un almacén de secretos dedicado (como HashiCorp Vault) en lugar de ser cargados directamente.
- **Control de Acceso (API):** El *endpoint* de inferencia se protegerá con una autenticación de tipo **API Key** simple, integrada en el *header* de la petición.
- **Minimización de Datos:** La implementación de K-Means en *Entrenamiento.py* opera solo sobre las 5 *features* numéricas. Los campos identificativos de texto (*'artist_name'*, *'track_id'*) se eliminan antes de la estandarización, garantizando la **minimización de datos** y reduciendo cualquier riesgo de privacidad.

Análisis de Costes Hipotéticos

El despliegue de nuestra arquitectura MLOps conlleva los siguientes costes operacionales, asumiendo un despliegue en una nube educativa:

Componente	Uso Estimado	Coste Hipotético Estimado (Mensual)
Cómputo (Airflow/Retraining)	1 VM de 8 vCPU / 32GB RAM (2 horas/semana)	20 €
Serving (API FastAPI)	Servicio Contenerizado (24/7, bajo tráfico, con auto escalado mínimo)	35 €
Almacenamiento (S3/GCS)	50 GB (Dataset, Logs, Modelos)	3 €
Monitorización (Grafana/Prometheus)	Despliegue en contenedor con uso mínimo de recursos	7 €
TOTAL (Estimado Mínimo)		65 €/mes

Este análisis muestra que, a pesar de la complejidad de la arquitectura MLOps, los costes de mantenimiento pueden mantenerse bajos en un entorno de bajo tráfico o académico, optimizando los recursos de cómputo para los procesos de *retraining* (que son esporádicos y no continuos). El principal coste operativo recae en el *Serving* continuo de la API.