

# Creando una Feature Store

(o cualquier otra herramienta para developers)

(Sencilla)

(Incremental)

(Con IA)

# ¿Quién soy y qué voy a contar?

Alejandro Vidal

- Fundador de mindmakers (edtech GenAI)
- Consultoría de GenAI
- Background: Psicología + Informática

## ¿Por qué estoy aquí?

- Muchos clientes **necesitan** una Feature Store (aunque no lo sepan).
- Pero no saben cómo empezar y las actuales necesitan refactorización o despliegues.
  - **Consultor empático:** no dejes en un equipo lo que tu no querrías mantener ;)
- Por eso comencé a desarrollar una librería *incremental* y *sencilla* para crear FS.
- ¿Cómo lo implementé? ¿Cómo usé IA Generativa para ello? Mejores prácticas para usar una FS.



# Modelos mentales y librerías

Desarrollar una librería es 50% programar y 50% diseñar.

Tu API debe seguir el modelo mental del usuario. Principio de la mínima sorpresa

Cuando una herramienta de desarrollo te encanta es porque encaja con tu modelo mental (y con el del problema).

```
@log_calls # Decorador
def my_function(x):
    return x + 1
```

```
def log_calls(fn):
    def wrapper(*args, **kwargs):
        print(f'Llamada a {fn.__name__} con {args} y {kwargs}')
        return fn(*args, **kwargs)

    return wrapper
```

```
def log_calls(fn):
    def wrapper(*args, **kwargs):
        print(f'Llamada a {fn.__name__} con {args} y {kwargs}')
        return None

    return wrapper
```

¿Por qué esperamos la izquierda?

Porque el modelo mental de un decorador es **modificar** la función.

Alejandro Vidal



# ¿Qué es una Feature Store?

## Lo que dicen los libros

Una Feature Store es un repositorio de features de manera que se puedan:

**reutilizar**: dos DS pueden usar la misma feature.

**administrar**: versionar, documentar, etc.

**materializar (aka. cachear)**: no volver a calcular una feature ya calculada.

## Lo que ocurre en la práctica

```
1 import pandas as pd  
2 df = pd.read_csv('data.csv')
```



# ¿Qué es una Feature Store?

## Lo que dicen los libros

Una Feature Store es un repositorio de features de manera que se puedan:

**reutilizar**: dos DS pueden usar la misma feature.

**administrar**: versionar, documentar, etc.

**materializar (aka. cachear)**: no volver a calcular una feature ya calculada.

## Lo que ocurre en la práctica

```
1 import pandas as pd  
2 df = pd.read_csv('data.csv')  
3  
4 df = df[ "blah" ].fillna( ... ) # ¿Esto es una feature?
```



# ¿Qué es una Feature Store?

## Lo que dicen los libros

Una Feature Store es un repositorio de features de manera que se puedan:

**reutilizar**: dos DS pueden usar la misma feature.

**administrar**: versionar, documentar, etc.

**materializar (aka. cachear)**: no volver a calcular una feature ya calculada.

## Lo que ocurre en la práctica

```
1 import pandas as pd
2 df = pd.read_csv('data.csv')
3
4 df = df["blah"].fillna( ... )
5
6 df["minor"] = df["age"] < 18 # ¿Esto es una feature?
```



# ¿Qué es una Feature Store?

## Lo que dicen los libros

Una Feature Store es un repositorio de features de manera que se puedan:

**reutilizar**: dos DS pueden usar la misma feature.

**administrar**: versionar, documentar, etc.

**materializar (aka. cachear)**: no volver a calcular una feature ya calculada.

## Lo que ocurre en la práctica

```
1 import pandas as pd
2 df = pd.read_csv('data.csv')
3 df = df["blah"].fillna( ... )
4 df["minor"] = df["age"] < 18
5
6 from sklearn.ensemble import RandomForestClassifier
7 from sklearn.model_selection import train_test_split
8 X = df.drop(columns=["target"])
9 y = df["target"]
10 X_train, X_test, y_train, y_test = train_test_split(X, y)
11 clf = RandomForestClassifier()
12 clf.fit(X_train, y_train)
```

Una vs 7 líneas de código. Las features se consideran parte del modelo de forma "intuitiva".



# Intuitivamente las features:

- Son "añadidos" al modelo en el "preprocesamiento".
- Son pequeñas (<1% del código) y abundantes.
- Están relacionadas por conocimiento de negocio.
- Tienden a imitar un modelo mental como este:

```
cosa_con_datos["nueva_feature"] = operacion(cosa_con_datos["otra_feature"],  
                                              cosa_con_datos["otra_mas"])  
  
cosa_con_datos.mutate("nueva_feature", cosa_que_calcula)  
  
add_feature(datos, "nueva_feature", cosa_que_calcula)
```

- O más técnicamente: mutación de un conjunto de datos añadiendo una nueva característica.
- Puede ser tabular o no (grafos, etc.)



# La mínima feature store

```
1 df = pd.read_csv('data.csv')
2
3 df = df["blah"].fillna( ... )
4
5 # Refactoriza para: documentar y reutilizar
6 # ¿Qué se te ocurre sencillo, en Python e *incremental*?
7 df["minor"] = df["age"] < 18
```



# La mínima feature store

```
1 df = pd.read_csv('data.csv')
2
3 df = df["blah"].fillna( ... )
4
5 def minor_feature(df): # Versión impura
6     """
7     Calcula si una persona es menor de edad en la variable `minor`.
8     """
9     df["minor"] = df["age"] < 18
10
11 minor_feature(df)
```



# La mínima feature store

```
1 df = pd.read_csv('data.csv')
2
3 df = df["blah"].fillna( ... )
4
5 def minor_feature(df): # Versión pura
6     """
7     Calcula si una persona es menor de edad en la variable `minor`.
8     """
9     return df["age"] < 18
10
11 df["minor"] = minor_feature(df)
```



# La mínima feature store

```
1 df = pd.read_csv('data.csv')
2
3 df = df["blah"].fillna( ... )
4
5 def minor_feature(age: pd.Series) → pd.Series: # Versión pura II
6     """
7         Calcula si una persona es menor de edad en la variable `minor`.
8     """
9     return age < 18
10
11 df["minor"] = minor_feature(df["age"])
```



# Decisiones de diseño

Tenéis que **investigar** (User Research) cómo vuestros usuarios **piensan** sobre el problema.

Vamos a usar **funciones**. Es la abstracción "natural" cuando quieras extraer una feature.

Las features tienen **nombre** (único).

Anda... ¡como las funciones!

Incremental: puedes migrar poco a poco.

Anda... ¡como las funciones!

No se asocia a una librería en particular para el cálculo (pandas, polars, ibis, etc.)

Anda... ¡como las funciones!

Pueden ser tablas o no (grafos, etc.) pero siempre hay un ID y un valor(es): key-value.





# Tres casos de uso de GenAI

Que no son programar

- Generador de "modelos mentales": APIs, nomenclatura, patrones de uso, ...
- Documentación asistida.

Language	files	comment	code
Python	37	2187	3797

Code/Doc ratio: 1.73

- "LLM Personas": ¿Es usable el framework dado un nivel de conocimiento?



# Generador de "modelos mentales"

Estoy creando una librería de python para hacer un feature store. Los aspectos más relevantes son:

- Agnóstica a la herramienta de cómputo (polars, pandas, spark, ...)
- Funciones como fábricas de features: cada feature tiene una función
- Decoración y anotaciones para añadir los metadatos. Genera algunos ejemplos de uso de esta librería.

```
from pyfeaturestore import feature, FeatureStore
from pyfeaturestore.types import Numeric, Categorical
from datetime import datetime

fs = FeatureStore()

@feature(
    name="edad_cliente",
    description="Edad del cliente en años",
    entity="cliente",
    value_type=Numeric(min_value=0, max_value=120)
)
def calcular_edad_cliente(data):
    return data['fecha_nacimiento'].map(lambda x: (datetime.now() - x).days // 365)

# ...
```

Tip: si tu librería ya está publicada cambia el nombre de las interfaces/librerías.



# LLM "Personas"

Compruebo si es incremental: puedes comenzar sin profundizar en los detalles del framework.

simple.py Current File

Basandote en el tutorial de `@simple.py` crea una feature del dataset de titanic y otra feature derivada de esta primera.

Claro, voy a crear dos features basadas en el dataset de Titanic: una feature inicial y otra derivada de la primera. Primero, vamos a definir una función para cargar el dataset de Titanic, y luego crearemos las features.

Aquí está el código con las nuevas features:

```
import polars as pl
import featurestore as fs

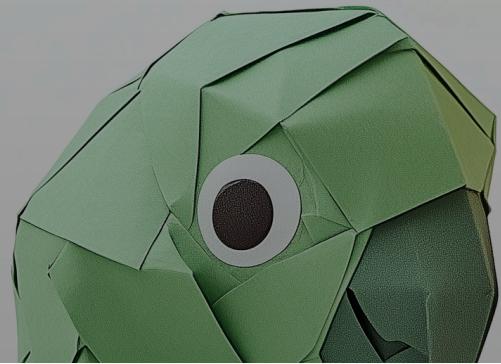
fs.config.interactive = True

def titanic_dataset():
    return pl.scan_csv("titanic.csv",
                       row_index_name="id")

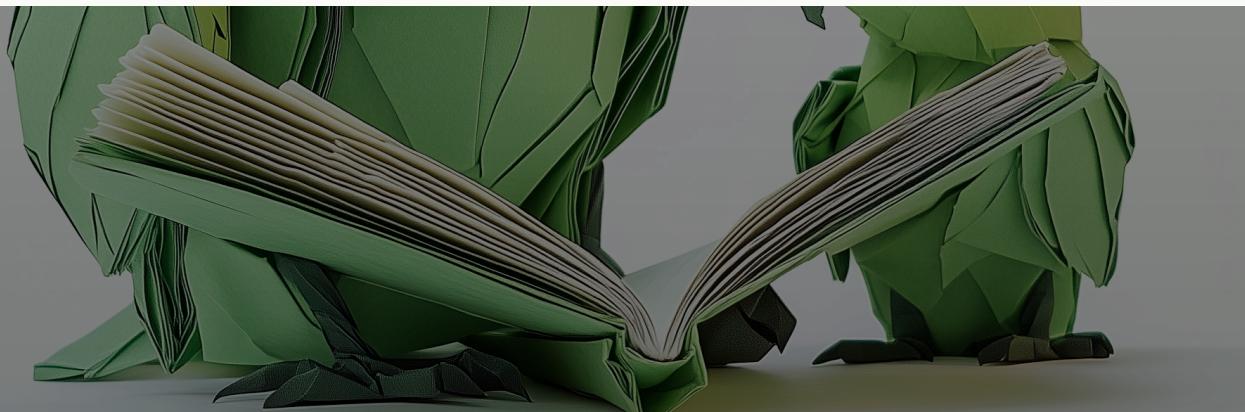
@fs.register_feature
def fare_per_person():
    """
    Calculate the fare per person for
    each passenger group.
    """

    df = titanic_dataset()
```





# Resultado final y aprendizajes



Alejandro Vida



# Features "dimensionales": tiempo, categorías, etc.

`simplefeaturestore` (nombre provisional) gestiona el tiempo (point in time features) como cualquier otro parámetro.

	user_id	event_name	category_name	product_id	event_time
1		read	notification	A	2023-08-25 12:23:20
2		favorite	notification	B	2023-08-25 19:55:06

```
1 from datetime import datetime
2
3 @fs.register_feature
4 def n_events(when: datetime) → ibis.Table:
5     """
6         How many events a user has made at a given point-in-time (PIT)
7     """
8     return (
9         con.table("events")
10        .filter(_.event_time ≤ when)
11        .group_by("user_id")
12        .agg(n_events=_ .count())
13        .select(["user_id", "n_events"])
14    )
```



# Features "dimensionales": tiempo, categorías, etc.

`simplefeaturestore` (nombre provisional) gestiona el tiempo (point in time features) como cualquier otro parámetro.

user_id	event_name	category_name	product_id	event_time
1	read	notification	A	2023-08-25 12:23:20
2	favorite	notification	B	2023-08-25 19:55:06

```
1 @fs.register_feature
2 def n_events(when: datetime) → ibis.Table:
3     """
4     ¿Metadatos?
5
6     Usa sintaxis estándares o las tuyas propias.
7
8     Maintainer:
9         John Smith <john.smith@example.com>
10        App Team
11
12        Status:
13            Production
14
15        Tags:
16            PII, sensitive, appteam
17        """
18        ...
```



# Materialización

Un aprendizaje muy importante ha sido: **hay que separar el código de materialización del código de features**. La mayoría de equipos pequeños sufren debido a no tener esta separación en el código. `simplefeaturestore` lo hace de forma natural.

```
1 # features/events.py (DS)
2 @fs.register_feature
3 def nFavorites(when: datetime):
4     """
5         How many likes a user has made at a given point-in-time (PIT).
6
7     `when` is the point-in-time (PIT) we want to evaluate the feature at.
8     """
9     df = events()
10
11     return (
12         df
13             .filter(pl.col("event_name") == "favorite")
14             .filter(pl.col("event_time") <= when)
15             .group_by("user_id")
16             .agg(nFavorites=pl.len())
17             .select("user_id", "nFavorites")
18     )
```



# Materialización

Un aprendizaje muy importante ha sido: **hay que separar el código de materialización del código de features**. La mayoría de equipos pequeños sufren debido a no tener esta separación en el código. `simplefeaturestore` lo hace de forma natural.

```
1 # features/events.py (DS)
2 @fs.register_feature
3 def nFavorites(when: datetime):
4     ...
5
6 # inferenceserver/main.py (XOps)
7 from featurestore.materializer.postgresql import PostgresMaterializer
8
9 materializer = PostgresMaterializer(
10     schema="feature_store",
11     conn=psycopg2.connect(
12         database="postgres",
13         user="postgres",
14         password="password",
15         host="localhost",
16     )
17 )
```



# Materialización

Un aprendizaje muy importante ha sido: **hay que separar el código de materialización del código de features**. La mayoría de equipos pequeños sufren debido a no tener esta separación en el código. `simplefeaturestore` lo hace de forma natural.

```
1 # features/events.py (DS)
2 @fs.register_feature
3 def nFavorites(when: datetime):
4     ...
5
6 # inferenceserver/main.py (XOps)
7 from featurestore.materializer.postgresql import PostgresMaterializer
8
9 materializer = PostgresMaterializer( ... )
10
11 from featurestore import fs
12
13 fs.add_decorator(materializer.decorator) # Todas las features se materializan
```



# Materialización

Un aprendizaje muy importante ha sido: **hay que separar el código de materialización del código de features**. La mayoría de equipos pequeños sufren debido a no tener esta separación en el código. `simplefeaturestore` lo hace de forma natural.

```
1 # features/events.py (DS)
2 @fs.register_feature
3 def nFavorites(when: datetime):
4     ...
5
6 # inferenceserver/main.py (XOps)
7 from featurestore.materializer.postgresql import PostgresMaterializer
8
9 materializer = PostgresMaterializer( ... )
10
11 from featurestore import fs
12
13 fs.add_decorator(materializer.decorator) # Todas las features se materializan
14
15 # Puedes tener distintas estrategias de materialización (velocidad, etc.)
16 # y aplicarlas a distintos módulos para separar responsabilidades entre
17 # equipos de dominio (producto A, producto B, etc.)
18 fs.add_decorator(materializerB.decorator, "features.another_module")
```



# Materialización

Un aprendizaje muy importante ha sido: **hay que separar el código de materialización del código de features**. La mayoría de equipos pequeños sufren debido a no tener esta separación en el código. `simplefeaturestore` lo hace de forma natural.

```
1 # features/events.py (DS)
2 @fs.register_feature
3 def nFavorites(when: datetime):
4     ...
5
6 # inferenceserver/main.py (XOps)
7 from featurestore.materializer.postgresql import PostgresMaterializer
8
9 materializer = PostgresMaterializer( ... )
10
11 from featurestore import fs
12
13 fs.add_decorator(materializer.decorator)
14
15 # inferenceserver/weekly_job.py (XOps)
16 from .main import materializer
17
18 # Self explanatory. Buena API.
19 materializer.materialize(nFavorites, when=datetime.now())
```

Alejandro Vidal



# Materialización y consistencia temporal

```
1 materializer = PostgresMaterializer( ... )
2
3 @fs.register_feature
4 def nFavorites(when: datetime, app_name: str) → ibis.Table:
5     ...
6
7 @fs.register_feature
8 def cohort(when: datetime, app_name: str) → ibis.Table:
9     ...
```



# Materialización y consistencia temporal

```
1 materializer = PostgresMaterializer( ... )
2
3 @fs.register_feature
4 def nFavorites(when: datetime, app_name: str):
5     ...
6
7 @fs.register_feature
8 def cohort(when: datetime, app_name: str):
9     ...
10
11 from featurestore.resolvers import Last, First
12
13 today = ...
14
15 (
16     fs.with_params(when=today, app_name="myapp") # PIT consistente
17     .get_features(nFavorites, nEvents)
18 )
```



# Realtime + laziness + decoración = Magia no mágica.

Si eliges las interfaces adecuadas los problemas se vuelven sencillos y queda un código "mágico" pero no mágico:

```
1 materializer.list_materializations(nFavorites)
2 # (Materialization(feature_name='__main__.nFavorites', params={'when': '2024-09-11 00:00:00.000000'}),
3 # Materialization(feature_name='__main__.nFavorites', params={'when': '2024-09-12 00:00:00.000000'}))
```



# Realtime + laziness + decoración = Magia no mágica.

Si eliges las interfaces adecuadas los problemas se vuelven sencillos y queda un código "mágico" pero no mágico:

```
1 # inferenceserver/main.py
2 from featurestore.resolvers import Last
3
4 def endpoint(request):
5     (
6         fs.with_params(
7             # En tiempo de inferencia no conozco cuál es el último valor materializado
8             # de cada feature (errores de ETL, latencias, etc.)
9             when=Last(),
10            # PIT consistente y será el último valor *consistente* de ambas features
11            app_name="myapp"
12        )
13        .get_features(nFavorites, nEvents)
14    )
```



# Realtime + laziness + decoración = Magia no mágica.

Si eliges las interfaces adecuadas los problemas se vuelven sencillos y queda un código "mágico" pero no mágico:

```
1 # inferenceserver/main.py
2 from featurestore.resolvers import Last
3
4 def endpoint(request):
5     (
6         fs.with_params(when=Last(), app_name="myapp")
7             .get_features(nFavorites, nEvents)
8             # Aquí se ve cómo la decoración es la abstracción "natural". El DS/XOps
9             # obtiene lo mismo que obtendría sin el decorador. En este caso un DF de
10            # polars. Podemos seguir filtrando... ;) ;como siempre!
11    )
```



# Realtime + laziness + decoración = Magia no mágica.

Si eliges las interfaces adecuadas los problemas se vuelven sencillos y queda un código "mágico" pero no mágico:

```
1 # inferenceserver/main.py
2 from featurestore.resolvers import Last
3
4 def endpoint(request):
5     (
6         fs.with_params(when=Last(), app_name="myapp")
7             .get_features(nFavorites, nEvents)
8             # Podemos seguir filtrando... ;) ¡como siempre!
9             .filter(pl.col("user_id") == request.user_id)
10    )
```



# Realtime + laziness + decoración = Magia no mágica.

Si eliges las interfaces adecuadas los problemas se vuelven sencillos y queda un código "mágico" pero no mágico:

```
1 # inferenceserver/main.py
2 from featurestore.resolvers import Last
3
4 def endpoint(request):
5     (
6         fs.with_params(when=Last(), app_name="myapp")
7             .get_features(nFavorites, nEvents)
8             .filter(pl.col("user_id") == request.user_id)
9             # Gracias a la magia del predicate pushdown y los DF lazy, esto es
10            # 100% equivalente a montar tu propia infraestructura de cacheo.
11            #
12            # Es decir, el filtro se hace en la BBDD/caché optimizado con índices
13            # siendo "python normal" y permitiendo dividir responsabilidades entre
14            # equipos de DS y XOps.
15     )
```



# Conclusiones

Las Feature Stores son más necesarias de lo que se les reconoce.

**¡Es el primer punto de fricción y problemas para muchos equipos!** Especialmente si no pueden separar el código de materialización del código de features.

**Si estás en la situación de "los servicios de data se caen y tenemos X BBDD/Redis/... responsabilidad de data"** es posible que vosotros estéis en esa situación. Añadid un FS lo antes posible.

La mayoría de FS actuales requieren refactorizaciones grandes y aprender su tipado, peculiaridades, etc. No es factible para los equipos que más lo necesitan (1-15 personas).

`simplefeaturestore` es agnóstica a la librería de cómputo y a la BBDD de materialización.

Lo más importante para mí: incremental y sencilla. Puedo aplicarla, enseñarla en mis clientes e irme con la confianza de que pueden continuar con su uso.

Las LLMs han sido fundamentales para diseñar correctamente esta librería.



# ¡Gracias!

- Si vas a implementar una FS
- Si quieras usar `simplefeaturestore`
- Si quieres aprender cómo usar GenAI para programar

Alejandro Vidal [alex@mindmake.rs](mailto:alex@mindmake.rs)

