MÓDULO 06: PIPELINES SKLEARN AVANZADOS

Custom Transformers, FeatureUnion y Prevención de Data Leakage

Guía MLOps v5.0: Senior Edition | DuqueOM | Noviembre 2025

⚙ MÓDULO 06: Pipelines Sklearn Avanzados

De Notebooks Experimentales a Código Production-Ready

"Un pipeline bien diseñado es la diferencia entre un experimento y un producto."

Duración	Teoría	Práctica
6-7 horas	25%	75%

Lo Que Lograrás en Este Módulo

1. **Construir** pipelines sklearn que encapsulan todo el preprocesamiento
2. **Crear** Custom Transformers reutilizables y testeables
3. **Evitar** Data Leakage con validación correcta
4. **Diseñar** pipelines modulares con ColumnTransformer y FeatureUnion

6.1 El Problema: Código de Entrenamiento vs Inferencia

```
╔═══════════════════════════════════════════════════════════════╗
║                  😱 EL ANTI-PATRÓN CLÁSICO                    ║
╚═══════════════════════════════════════════════════════════════╝

ENTRENAMIENTO (notebook):
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
encoder = OneHotEncoder()
X_train_encoded = encoder.fit_transform(X_cat)
X_train_final = np.hstack([X_train_scaled, X_train_encoded])
model.fit(X_train_final, y_train)
joblib.dump(model, "model.pkl")
# ¿Y el scaler? ¿Y el encoder?

INFERENCIA (otro archivo, 3 meses después):
model = joblib.load("model.pkl")
# ???  ¿Cómo preproceso los nuevos datos? 😱
# ¿Qué columnas? ¿Qué orden? ¿Qué scaler?

╔═══════════════════════════════════════════════════════════════╗
║                  LA SOLUCIÓN: PIPELINE UNIFICADO              ║
╚═══════════════════════════════════════════════════════════════╝

pipeline = Pipeline([
    ('preprocessor', ColumnTransformer([...])),
    ('model', RandomForestClassifier())
])
pipeline.fit(X_train, y_train)
joblib.dump(pipeline, "pipeline.pkl")

# Inferencia (simple y segura):
pipeline = joblib.load("pipeline.pkl")
predictions = pipeline.predict(X_new)  # ¡Ya sabe cómo preprocesar!
```

6.2 Anatomía de un Pipeline Profesional

Estructura Básica

```python
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.ensemble import RandomForestClassifier

# Definir columnas
NUMERICAL_FEATURES = ['CreditScore', 'Age', 'Tenure', 'Balance', 'NumOfProducts', 'EstimatedSalary']
CATEGORICAL_FEATURES = ['Geography', 'Gender']
BINARY_FEATURES = ['HasCrCard', 'IsActiveMember']

# ═══════════════════════════════════════════════════════════
# PREPROCESADOR: ColumnTransformer
# ═══════════════════════════════════════════════════════════
preprocessor = ColumnTransformer(
    transformers=[
        # (nombre, transformer, columnas)
        ('num', Pipeline([
            ('imputer', SimpleImputer(strategy='median')),
            ('scaler', StandardScaler())
        ]), NUMERICAL_FEATURES),

        ('cat', Pipeline([
            ('imputer', SimpleImputer(strategy='constant', fill_value='missing')),
            ('encoder', OneHotEncoder(handle_unknown='ignore', sparse_output=False))
        ]), CATEGORICAL_FEATURES),

        ('bin', 'passthrough', BINARY_FEATURES),  # Sin transformación
    ],
    remainder='drop',  # Ignorar columnas no especificadas
    verbose_feature_names_out=False,
)

# ═══════════════════════════════════════════════════════════
# PIPELINE COMPLETO
# ═══════════════════════════════════════════════════════════
pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('classifier', RandomForestClassifier(
        n_estimators=100,
        max_depth=10,
        random_state=42,
        class_weight='balanced',
        n_jobs=-1,
    ))
])

# ═══════════════════════════════════════════════════════════
# USO
# ═══════════════════════════════════════════════════════════
# Entrenamiento
pipeline.fit(X_train, y_train)

# Predicción (incluye preprocesamiento automáticamente)
predictions = pipeline.predict(X_test)
probabilities = pipeline.predict_proba(X_test)[:, 1]

# Guardar TODO junto
import joblib
joblib.dump(pipeline, 'models/pipeline.pkl')
```

Visualización del Pipeline

```python
from sklearn import set_config
set_config(display='diagram')

# En Jupyter, esto muestra el pipeline gráficamente
pipeline
```

```
╔═══════════════════════════════════════════════════════════════╗
║                        PIPELINE VISUAL                        ║
╠═══════════════════════════════════════════════════════════════╣
║                                                               ║
║  Pipeline                                                     ║
║  ├── preprocessor: ColumnTransformer                          ║
║  │   ├── num: Pipeline                                        ║
║  │   │   ├── imputer: SimpleImputer(strategy='median')        ║
║  │   │   └── scaler: StandardScaler()                         ║
║  │   ├── cat: Pipeline                                        ║
║  │   │   ├── imputer: SimpleImputer(strategy='constant')      ║
║  │   │   └── encoder: OneHotEncoder(handle_unknown='ignore')  ║
║  │   └── bin: passthrough                                     ║
║  └── classifier: RandomForestClassifier(class_weight='balanced')
║                                                               ║
╚═══════════════════════════════════════════════════════════════╝
```

6.3 Custom Transformers

¿Por Qué Custom Transformers?

- Encapsular lógica de feature engineering
- Reutilizar entre proyectos
- Testear unitariamente
- Mantener todo en el pipeline

Template de Custom Transformer

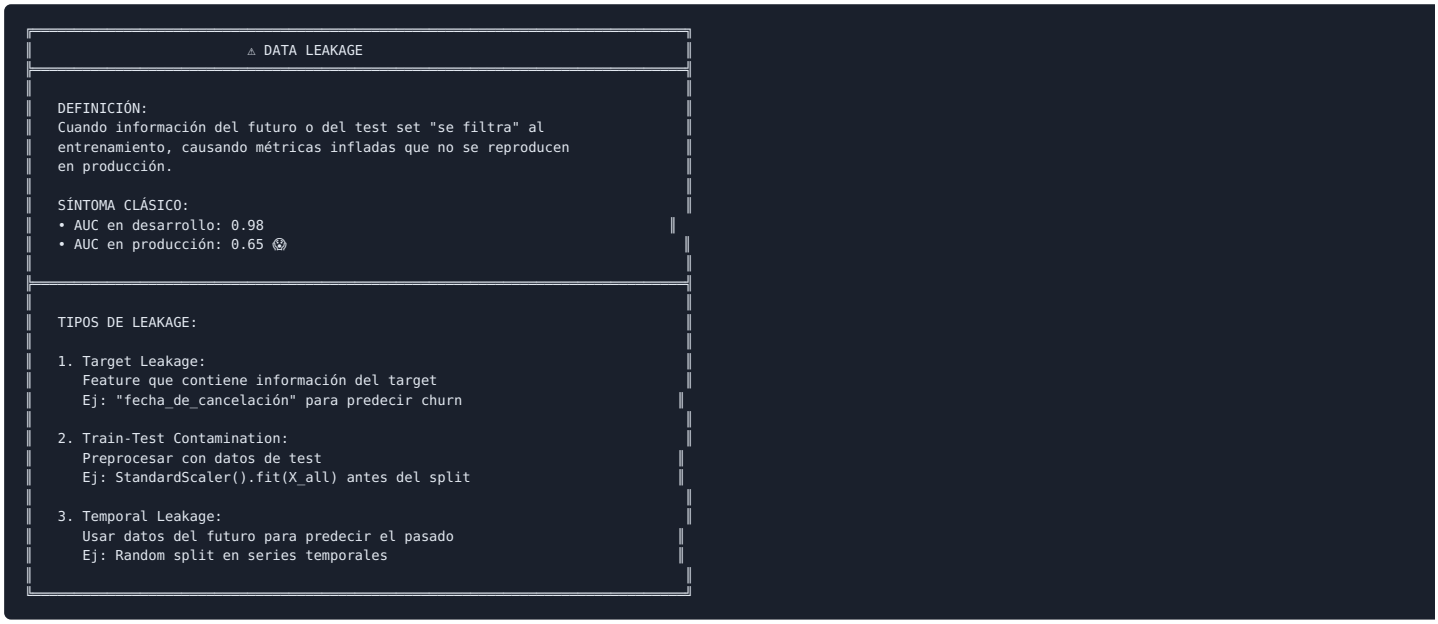
```python
from sklearn.base import BaseEstimator, TransformerMixin
from typing import Optional
import pandas as pd
import numpy as np

class FeatureEngineer(BaseEstimator, TransformerMixin):
    """
    Custom transformer para feature engineering.

    Sigue la API de sklearn:
    - fit(X, y=None) → self
    - transform(X) → X_transformed
    - fit_transform(X, y=None) → X_transformed

    Attributes:
        feature_names_out_: List[str] - Nombres de features después de transform
    """

    def __init__(
        self,
        add_ratios: bool = True,
        add_interactions: bool = False,
    ):
        """
        Args:
            add_ratios: Si añadir features de ratio (Balance/Salary, etc.)
            add_interactions: Si añadir interacciones entre features
        """
        self.add_ratios = add_ratios
        self.add_interactions = add_interactions

    def fit(self, X: pd.DataFrame, y: Optional[pd.Series] = None) -> "FeatureEngineer":
        """
        Aprende parámetros necesarios del dataset de entrenamiento.

        En este caso, solo guardamos los nombres de columnas para validación.
        """
        # Validar que X es DataFrame
        if not isinstance(X, pd.DataFrame):
            raise TypeError("X debe ser un pandas DataFrame")

        # Guardar columnas esperadas
        self.feature_names_in_ = list(X.columns)

        # Calcular nombres de salida
        self.feature_names_out_ = self._compute_feature_names(X)

        return self

    def transform(self, X: pd.DataFrame) -> pd.DataFrame:
        """
        Aplica las transformaciones aprendidas en fit.
        """
        # Validar que fit fue llamado
        if not hasattr(self, 'feature_names_in_'):
            raise RuntimeError("Debes llamar fit() antes de transform()")

        # Copiar para no modificar original
        X_transformed = X.copy()

        # Añadir ratios
        if self.add_ratios:
            X_transformed = self._add_ratio_features(X_transformed)

        # Añadir interacciones
        if self.add_interactions:
            X_transformed = self._add_interaction_features(X_transformed)

        return X_transformed

    def _add_ratio_features(self, X: pd.DataFrame) -> pd.DataFrame:
        """Añade features de ratio."""
        # Balance por producto
        if 'Balance' in X.columns and 'NumOfProducts' in X.columns:
            X['BalancePerProduct'] = X['Balance'] / (X['NumOfProducts'] + 1)

        # Balance / Salario
        if 'Balance' in X.columns and 'EstimatedSalary' in X.columns:
            X['BalanceSalaryRatio'] = X['Balance'] / (X['EstimatedSalary'] + 1)

        # Tenure / Age
        if 'Tenure' in X.columns and 'Age' in X.columns:
            X['TenureAgeRatio'] = X['Tenure'] / (X['Age'] + 1)

        return X

    def _add_interaction_features(self, X: pd.DataFrame) -> pd.DataFrame:
        """Añade features de interacción."""
        if 'Age' in X.columns and 'Balance' in X.columns:
            X['Age_x_Balance'] = X['Age'] * X['Balance']
        return X

    def _compute_feature_names(self, X: pd.DataFrame) -> list:
        """Calcula los nombres de features de salida."""
        names = list(X.columns)

        if self.add_ratios:
            if 'Balance' in X.columns and 'NumOfProducts' in X.columns:
                names.append('BalancePerProduct')
            if 'Balance' in X.columns and 'EstimatedSalary' in X.columns:
                names.append('BalanceSalaryRatio')
            if 'Tenure' in X.columns and 'Age' in X.columns:
                names.append('TenureAgeRatio')

        if self.add_interactions:
            if 'Age' in X.columns and 'Balance' in X.columns:
                names.append('Age_x_Balance')

        return names

    def get_feature_names_out(self, input_features: Optional[list] = None) -> np.ndarray:
        """Retorna nombres de features de salida (API sklearn 1.0+)."""
        return np.array(self.feature_names_out_)
```

Ejemplo: TargetEncoder Custom

```python
from sklearn.base import BaseEstimator, TransformerMixin
import pandas as pd
import numpy as np

class TargetEncoder(BaseEstimator, TransformerMixin):
    """
    Target Encoder que evita data leakage usando leave-one-out.

    Para cada categoría, calcula la media del target (con smoothing).
    """

    def __init__(self, smoothing: float = 10.0, min_samples: int = 5):
        self.smoothing = smoothing
        self.min_samples = min_samples

    def fit(self, X: pd.DataFrame, y: pd.Series) -> "TargetEncoder":
        """Calcula encodings para cada categoría."""
        self.encodings_ = {}
        self.global_mean_ = y.mean()

        for col in X.columns:
            # Calcular stats por categoría
            stats = pd.DataFrame({
                'category': X[col],
                'target': y
            }).groupby('category').agg({
                'target': ['mean', 'count']
            })
            stats.columns = ['mean', 'count']

            # Smoothing: blend con media global
            # encoding = (count * mean + smoothing * global_mean) / (count + smoothing)
            smoothed = (
                stats['count'] * stats['mean'] +
                self.smoothing * self.global_mean_
            ) / (stats['count'] + self.smoothing)

            # Para categorías con pocas muestras, usar global mean
            smoothed[stats['count'] < self.min_samples] = self.global_mean_

            self.encodings_[col] = smoothed.to_dict()

        return self

    def transform(self, X: pd.DataFrame) -> pd.DataFrame:
        """Aplica target encoding."""
        X_encoded = X.copy()

        for col in X.columns:
            X_encoded[col] = X[col].map(self.encodings_.get(col, {}))
            # Categorías no vistas → global mean
            X_encoded[col].fillna(self.global_mean_, inplace=True)

        return X_encoded
```

6.4 Prevención de Data Leakage

¿Qué es Data Leakage?



Errores Comunes y Soluciones

```python
# ═══════════════════════════════════════════════════════════════
# ✗ ERROR 1: Escalar ANTES del split
# ═══════════════════════════════════════════════════════════════

# MAL
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)  # ← Usa estadísticas de TODO el dataset
X_train, X_test = train_test_split(X_scaled, ...)

# BIEN
X_train, X_test = train_test_split(X, ...)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)  # Solo train
X_test_scaled = scaler.transform(X_test)  # Solo transform

# MEJOR (Pipeline lo hace automáticamente correcto)
pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('model', LogisticRegression())
])
pipeline.fit(X_train, y_train)  # scaler.fit solo ve X_train

# ═══════════════════════════════════════════════════════════════
# ✗ ERROR 2: Feature selection con todo el dataset
# ═══════════════════════════════════════════════════════════════

# MAL
from sklearn.feature_selection import SelectKBest
selector = SelectKBest(k=10)
X_selected = selector.fit_transform(X, y)  # ← Leakage
X_train, X_test = train_test_split(X_selected, ...)

# BIEN (dentro del pipeline)
pipeline = Pipeline([
    ('selector', SelectKBest(k=10)),
    ('model', RandomForestClassifier())
])
pipeline.fit(X_train, y_train)

# ═══════════════════════════════════════════════════════════════
# ✗ ERROR 3: Imputar con media de todo el dataset
# ═══════════════════════════════════════════════════════════════

# MAL
X['Age'].fillna(X['Age'].mean(), inplace=True)  # Media de TODO
X_train, X_test = train_test_split(X, ...)

# BIEN
X_train, X_test = train_test_split(X, ...)
train_mean = X_train['Age'].mean()
X_train['Age'].fillna(train_mean, inplace=True)
X_test['Age'].fillna(train_mean, inplace=True)

# ═══════════════════════════════════════════════════════════════
# ✗ ERROR 4: Target encoding fuera del pipeline
# ═══════════════════════════════════════════════════════════════

# MAL
encoder = TargetEncoder()
X['Geography'] = encoder.fit_transform(X[['Geography']], y)  # ← Leakage
X_train, X_test = train_test_split(X, ...)

# BIEN (Target encoding DEBE estar en el pipeline)
pipeline = Pipeline([
    ('encoder', TargetEncoder()),  # fit solo con train
    ('model', RandomForestClassifier())
])
```

Cross-Validation Correcta

```python
from sklearn.model_selection import cross_val_score, StratifiedKFold

# El pipeline COMPLETO debe ir dentro del CV
# Así el preprocesamiento se re-fitea en cada fold

pipeline = Pipeline([
    ('preprocessor', ColumnTransformer([...])),
    ('model', RandomForestClassifier())
])

cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

# Esto es correcto: el pipeline se fitea 5 veces, cada vez solo con train fold
scores = cross_val_score(pipeline, X, y, cv=cv, scoring='roc_auc')
print(f"AUC: {scores.mean():.4f} (+/- {scores.std()*2:.4f})")
```

Validación Temporal (Time Series)

```python
from sklearn.model_selection import TimeSeriesSplit

# Para datos con componente temporal, NUNCA usar random split
# El futuro no puede predecir el pasado

tscv = TimeSeriesSplit(n_splits=5)

for train_idx, test_idx in tscv.split(X):
    X_train, X_test = X.iloc[train_idx], X.iloc[test_idx]
    y_train, y_test = y.iloc[train_idx], y.iloc[test_idx]

    # Verificar que test siempre es después de train
    print(f"Train: {X_train.index.min()} - {X_train.index.max()}")
    print(f"Test:  {X_test.index.min()} - {X_test.index.max()}")
```

6.5 Pipeline Completo para BankChurn

```python
# src/bankchurn/pipeline.py
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.ensemble import RandomForestClassifier
from sklearn.base import BaseEstimator, TransformerMixin
import pandas as pd
import numpy as np
from typing import List, Optional

class FeatureEngineer(BaseEstimator, TransformerMixin):
    """Feature engineering para BankChurn."""

    def __init__(self, add_ratios: bool = True):
        self.add_ratios = add_ratios

    def fit(self, X: pd.DataFrame, y: Optional[pd.Series] = None) -> "FeatureEngineer":
        self.feature_names_in_ = list(X.columns)
        return self

    def transform(self, X: pd.DataFrame) -> pd.DataFrame:
        X = X.copy()
        if self.add_ratios:
            X['BalancePerProduct'] = X['Balance'] / (X['NumOfProducts'].clip(lower=1))
            X['BalanceSalaryRatio'] = X['Balance'] / (X['EstimatedSalary'].clip(lower=1))
        return X

    def get_feature_names_out(self, input_features=None):
        names = list(self.feature_names_in_)
        if self.add_ratios:
            names.extend(['BalancePerProduct', 'BalanceSalaryRatio'])
        return np.array(names)


def build_pipeline(
    numerical_features: List[str],
    categorical_features: List[str],
    binary_features: List[str],
    model_params: Optional[dict] = None,
) -> Pipeline:
    """
    Construye el pipeline completo de BankChurn.

    Args:
        numerical_features: Lista de features numéricas
        categorical_features: Lista de features categóricas
        binary_features: Lista de features binarias (0/1)
        model_params: Parámetros para RandomForestClassifier

    Returns:
        Pipeline sklearn completo
    """
    model_params = model_params or {
        'n_estimators': 100,
        'max_depth': 10,
        'random_state': 42,
        'class_weight': 'balanced',
        'n_jobs': -1,
    }

    # Preprocessor para features originales
    preprocessor = ColumnTransformer(
        transformers=[
            ('num', Pipeline([
                ('imputer', SimpleImputer(strategy='median')),
                ('scaler', StandardScaler()),
            ]), numerical_features),

            ('cat', Pipeline([
                ('imputer', SimpleImputer(strategy='constant', fill_value='Unknown')),
                ('encoder', OneHotEncoder(handle_unknown='ignore', sparse_output=False)),
            ]), categorical_features),

            ('bin', 'passthrough', binary_features),
        ],
        remainder='drop',
    )

    # Pipeline completo
    pipeline = Pipeline([
        ('features', FeatureEngineer(add_ratios=True)),
        ('preprocessor', preprocessor),
        ('classifier', RandomForestClassifier(**model_params)),
    ])

    return pipeline


# ═══════════════════════════════════════════════════════════════
# USO
# ═══════════════════════════════════════════════════════════════
if __name__ == "__main__":
    from bankchurn.config import load_config
    import joblib

    config = load_config("configs/config.yaml")

    pipeline = build_pipeline(
        numerical_features=config.features.numerical,
        categorical_features=config.features.categorical,
        binary_features=config.features.binary,
        model_params=config.model.model_dump(),
    )

    # Entrenar
    pipeline.fit(X_train, y_train)

    # Guardar
    joblib.dump(pipeline, config.paths.model_output)
```

6.6 Ejercicio Integrador

Construye Tu Pipeline

1. **Identifica** las columnas de tu dataset por tipo
2. **Diseña** el ColumnTransformer

3. **Crea** al menos 1 Custom Transformer
4. **Verifica** que no hay data leakage

Checklist de Verificación

```
ESTRUCTURA:
[ ] Pipeline usa ColumnTransformer para diferentes tipos de features
[ ] Preprocesamiento encapsulado (no se hace fuera del pipeline)
[ ] Pipeline guardable con joblib

CUSTOM TRANSFORMERS:
[ ] Al menos 1 transformer custom implementado
[ ] Transformer sigue API sklearn (fit/transform)
[ ] Transformer tiene get_feature_names_out()

DATA LEAKAGE:
[ ] fit() solo se llama en train data
[ ] Cross-validation usa el pipeline completo
[ ] No hay preprocesamiento antes del split
```

6.7 Autoevaluación

1. ¿Por qué el pipeline debe incluir el preprocesamiento y el modelo juntos?
2. ¿Cuál es la diferencia entre `fit_transform` y `transform`?
3. ¿Por qué `handle_unknown='ignore'` es importante en OneHotEncoder?
4. ¿Cómo verificarías que no hay data leakage en tu pipeline?

Siguiente Paso

Con pipelines robustos, es hora de **trackear experimentos** profesionalmente.

Ir a Módulo 07: Experiment Tracking →

Módulo 06 completado. Tu pipeline ahora es production-ready.