MÓDULO 08: TESTING PARA ML

Unit, Integration, Data Tests y Model Tests

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

MÓDULO 08: Testing para ML

Sin Tests No Hay Deployment

“Un modelo sin tests es una bomba de tiempo esperando explotar en producción.”

Duración	Teoría	Práctica
5-6 horas	20%	80%

La Pirámide de Testing para ML

```
╔════════════════════════════════════════════════════════════════════════════════╗
║                      PIRÁMIDE DE TESTING PARA ML                               ║
╠════════════════════════════════════════════════════════════════════════════════╣
║                                                                                ║
║                            ┌─────────────┐                                     ║
║                            │   E2E       │  ← Pocos, costosos, lentos          ║
║                            │  (API+ML)   │     "¿El sistema completo funciona?"║
║                          ┌─┴─────────────┴─┐                                   ║
║                          │  Integration    │  ← Pipeline completo              ║
║                          │ (Data→Predict)  │     "¿Los componentes conectan?"  ║
║                      ┌───┴─────────────────┴───┐                               ║
║                      │      Model Tests        │  ← Performance, Fairness      ║
║                      │   (Métricas, Drift)     │     "¿El modelo es bueno?"    ║
║                  ┌───┴─────────────────────────┴───┐                           ║
║                  │          Data Tests             │  ← Great Expectations     ║
║                  │   (Schema, Nulls, Ranges)       │     "¿Los datos son       ║
║              ┌───┴─────────────────────────────────┴───┐  válidos?"            ║
║              │              Unit Tests                 │  ← Muchos, rápidos    ║
║              │   (Funciones, Transformers, Utils)      │     "¿La función      ║
║              └─────────────────────────────────────────┘     funciona?"        ║
║                                                                                ║
║   DISTRIBUCIÓN RECOMENDADA:                                                    ║
║   • 60% Unit Tests         (< 1 segundo cada uno)                              ║
║   • 20% Data Tests         (validación de schema)                              ║
║   • 10% Model Tests        (métricas mínimas)                                  ║
║   • 7%  Integration Tests  (pipeline completo)                                 ║
║   • 3%  E2E Tests          (API funcionando)                                   ║
║                                                                                ║
╚════════════════════════════════════════════════════════════════════════════════╝
```

8.1 Unit Tests: La Base de Todo

Estructura de Tests

```
tests/
├── conftest.py              # Fixtures compartidos
├── unit/
│   ├── test_config.py
│   ├── test_preprocessing.py
│   ├── test_features.py
│   └── test_training.py
├── integration/
│   └── test_pipeline.py
├── data/
│   └── test_data_quality.py
└── model/
    └── test_model_performance.py
```

conftest.py: Fixtures Reutilizables

```python
# tests/conftest.py
import pytest
import pandas as pd
import numpy as np
from pathlib import Path

@pytest.fixture
def sample_data() -> pd.DataFrame:
    """DataFrame de ejemplo para tests."""
    np.random.seed(42)
    n = 100
    return pd.DataFrame({
        'CreditScore': np.random.randint(300, 850, n),
        'Age': np.random.randint(18, 80, n),
        'Tenure': np.random.randint(0, 15, n),
        'Balance': np.random.uniform(0, 250000, n),
        'NumOfProducts': np.random.randint(1, 5, n),
        'HasCrCard': np.random.randint(0, 2, n),
        'IsActiveMember': np.random.randint(0, 2, n),
        'EstimatedSalary': np.random.uniform(20000, 200000, n),
        'Geography': np.random.choice(['France', 'Germany', 'Spain'], n),
        'Gender': np.random.choice(['Male', 'Female'], n),
        'Exited': np.random.randint(0, 2, n),
    })

@pytest.fixture
def sample_X(sample_data) -> pd.DataFrame:
    """Features sin target."""
    return sample_data.drop(columns=['Exited'])

@pytest.fixture
def sample_y(sample_data) -> pd.Series:
    """Target."""
    return sample_data['Exited']

@pytest.fixture
def config():
    """Configuración de test."""
    from bankchurn.config import TrainingConfig
    return TrainingConfig(
        model={"n_estimators": 10, "max_depth": 3},  # Pequeño para tests rápidos
        features={
            "numerical": ['CreditScore', 'Age', 'Balance'],
            "categorical": ['Geography', 'Gender'],
            "binary": ['HasCrCard'],
        }
    )

@pytest.fixture
def trained_pipeline(sample_X, sample_y, config):
    """Pipeline ya entrenado."""
    from bankchurn.pipeline import build_pipeline
    pipeline = build_pipeline(
        numerical_features=config.features.numerical,
        categorical_features=config.features.categorical,
        binary_features=config.features.binary,
        model_params={"n_estimators": 10, "max_depth": 3, "random_state": 42},
    )
    pipeline.fit(sample_X, sample_y)
    return pipeline
```

Unit Tests para Config

```python
# tests/unit/test_config.py
import pytest
from pydantic import ValidationError
from bankchurn.config import ModelConfig, TrainingConfig

class TestModelConfig:
    """Tests para ModelConfig."""

    def test_default_values(self):
        """Config con valores por defecto debe ser válida."""
        config = ModelConfig()
        assert config.n_estimators == 100
        assert config.random_state == 42

    def test_valid_custom_values(self):
        """Config con valores custom válidos."""
        config = ModelConfig(n_estimators=200, max_depth=15)
        assert config.n_estimators == 200
        assert config.max_depth == 15

    def test_invalid_n_estimators_raises(self):
        """n_estimators < 10 debe fallar."""
        with pytest.raises(ValidationError) as exc_info:
            ModelConfig(n_estimators=5)
        assert "n_estimators" in str(exc_info.value)

    def test_invalid_max_depth_raises(self):
        """max_depth negativo debe fallar."""
        with pytest.raises(ValidationError):
            ModelConfig(max_depth=-1)

    @pytest.mark.parametrize("n_estimators,expected", [
        (10, 10),
        (100, 100),
        (1000, 1000),
    ])
    def test_valid_n_estimators_range(self, n_estimators, expected):
        """n_estimators válido en diferentes rangos."""
        config = ModelConfig(n_estimators=n_estimators)
        assert config.n_estimators == expected
```

Unit Tests para Preprocessing

```python
# tests/unit/test_preprocessing.py
import pytest
import pandas as pd
import numpy as np
from bankchurn.pipeline import FeatureEngineer

class TestFeatureEngineer:
    """Tests para FeatureEngineer transformer."""

    def test_fit_returns_self(self, sample_X):
        """fit() debe retornar self."""
        fe = FeatureEngineer()
        result = fe.fit(sample_X)
        assert result is fe

    def test_transform_adds_ratio_features(self, sample_X):
        """transform() debe añadir features de ratio."""
        fe = FeatureEngineer(add_ratios=True)
        fe.fit(sample_X)
        result = fe.transform(sample_X)

        assert 'BalancePerProduct' in result.columns
        assert 'BalanceSalaryRatio' in result.columns

    def test_transform_without_ratios(self, sample_X):
        """add_ratios=False no debe añadir features."""
        original_cols = set(sample_X.columns)
        fe = FeatureEngineer(add_ratios=False)
        fe.fit(sample_X)
        result = fe.transform(sample_X)

        assert set(result.columns) == original_cols

    def test_transform_handles_zero_division(self, sample_X):
        """Debe manejar división por cero sin errores."""
        sample_X = sample_X.copy()
        sample_X['NumOfProducts'] = 0  # Forzar división por cero

        fe = FeatureEngineer(add_ratios=True)
        fe.fit(sample_X)
        result = fe.transform(sample_X)

        # No debe haber inf o nan
        assert not np.isinf(result['BalancePerProduct']).any()

    def test_transform_before_fit_raises(self, sample_X):
        """transform() sin fit() debe fallar."""
        fe = FeatureEngineer()
        with pytest.raises(RuntimeError, match="fit"):
            fe.transform(sample_X)

    def test_get_feature_names_out(self, sample_X):
        """get_feature_names_out() debe retornar nombres correctos."""
        fe = FeatureEngineer(add_ratios=True)
        fe.fit(sample_X)
        names = fe.get_feature_names_out()

        assert 'BalancePerProduct' in names
        assert len(names) == len(sample_X.columns) + 2  # 2 ratios añadidos
```

8.2 Data Tests: Validar Datos de Entrada

Con Great Expectations (Recomendado)

```python
# tests/data/test_data_quality.py
import pytest
import pandas as pd
import great_expectations as gx

class TestDataQuality:
    """Tests de calidad de datos."""

    @pytest.fixture
    def expectations_suite(self):
        """Define expectativas para el dataset."""
        return {
            "CreditScore": {"min": 300, "max": 850, "not_null": True},
            "Age": {"min": 18, "max": 120, "not_null": True},
            "Balance": {"min": 0, "not_null": True},
            "Geography": {"values": ["France", "Germany", "Spain"], "not_null": True},
            "Gender": {"values": ["Male", "Female"], "not_null": True},
            "Exited": {"values": [0, 1], "not_null": True},
        }

    def test_no_null_values_in_critical_columns(self, sample_data):
        """Columnas críticas no deben tener nulos."""
        critical_cols = ['CreditScore', 'Age', 'Geography', 'Exited']
        for col in critical_cols:
            null_count = sample_data[col].isnull().sum()
            assert null_count == 0, f"{col} tiene {null_count} nulos"

    def test_credit_score_range(self, sample_data):
        """CreditScore debe estar entre 300 y 850."""
        assert sample_data['CreditScore'].min() >= 300
        assert sample_data['CreditScore'].max() <= 850

    def test_age_range(self, sample_data):
        """Age debe ser razonable."""
        assert sample_data['Age'].min() >= 18
        assert sample_data['Age'].max() <= 120

    def test_balance_non_negative(self, sample_data):
        """Balance no puede ser negativo."""
        assert (sample_data['Balance'] >= 0).all()

    def test_target_is_binary(self, sample_data):
        """Target debe ser binario."""
        unique_values = set(sample_data['Exited'].unique())
        assert unique_values.issubset({0, 1})

    def test_geography_valid_categories(self, sample_data):
        """Geography solo debe tener categorías válidas."""
        valid = {'France', 'Germany', 'Spain'}
        actual = set(sample_data['Geography'].unique())
        assert actual.issubset(valid), f"Categorías inválidas: {actual - valid}"

    def test_no_duplicate_rows(self, sample_data):
        """No debe haber filas duplicadas completas."""
        n_duplicates = sample_data.duplicated().sum()
        assert n_duplicates == 0, f"Encontradas {n_duplicates} filas duplicadas"

    def test_data_schema(self, sample_data):
        """Schema del DataFrame debe ser correcto."""
        expected_schema = {
            'CreditScore': 'int64',
            'Age': 'int64',
            'Balance': 'float64',
            'Geography': 'object',
            'Exited': 'int64',
        }
        for col, dtype in expected_schema.items():
            assert col in sample_data.columns, f"Falta columna {col}"
            assert str(sample_data[col].dtype) == dtype, \
                f"{col} tiene dtype {sample_data[col].dtype}, esperado {dtype}"
```

Con Pandera (Alternativa Más Simple)

```python
# tests/data/test_schema.py
import pandera as pa
from pandera import Column, Check, DataFrameSchema
import pytest
import pandas as pd

# Definir schema
churn_schema = DataFrameSchema({
    "CreditScore": Column(int, Check.in_range(300, 850)),
    "Age": Column(int, Check.in_range(18, 120)),
    "Tenure": Column(int, Check.ge(0)),
    "Balance": Column(float, Check.ge(0)),
    "NumOfProducts": Column(int, Check.in_range(1, 4)),
    "HasCrCard": Column(int, Check.isin([0, 1])),
    "IsActiveMember": Column(int, Check.isin([0, 1])),
    "EstimatedSalary": Column(float, Check.ge(0)),
    "Geography": Column(str, Check.isin(["France", "Germany", "Spain"])),
    "Gender": Column(str, Check.isin(["Male", "Female"])),
    "Exited": Column(int, Check.isin([0, 1])),
})

def test_data_matches_schema(sample_data):
    """Dataset debe cumplir el schema."""
    validated_df = churn_schema.validate(sample_data)
    assert validated_df is not None
```

8.3 Model Tests: Validar Performance

```python
# tests/model/test_model_performance.py
import pytest
import numpy as np
from sklearn.metrics import roc_auc_score, precision_score, recall_score

class TestModelPerformance:
    """Tests de performance del modelo."""

    def test_model_predictions_shape(self, trained_pipeline, sample_X):
        """Predicciones deben tener shape correcto."""
        predictions = trained_pipeline.predict(sample_X)
        assert predictions.shape == (len(sample_X),)

    def test_model_probabilities_sum_to_one(self, trained_pipeline, sample_X):
        """Probabilidades deben sumar 1."""
        probas = trained_pipeline.predict_proba(sample_X)
        row_sums = probas.sum(axis=1)
        np.testing.assert_array_almost_equal(row_sums, 1.0)

    def test_model_probabilities_range(self, trained_pipeline, sample_X):
        """Probabilidades deben estar entre 0 y 1."""
        probas = trained_pipeline.predict_proba(sample_X)
        assert (probas >= 0).all()
        assert (probas <= 1).all()

    def test_minimum_auc_threshold(self, trained_pipeline, sample_X, sample_y):
        """AUC debe superar umbral mínimo."""
        y_proba = trained_pipeline.predict_proba(sample_X)[:, 1]
        auc = roc_auc_score(sample_y, y_proba)

        # En tests, usamos umbral bajo porque es data sintética
        # En producción, este umbral debe ser más alto (e.g., 0.75)
        MIN_AUC = 0.50  # Mejor que random
        assert auc >= MIN_AUC, f"AUC {auc:.4f} está por debajo del umbral {MIN_AUC}"

    def test_predictions_not_all_same_class(self, trained_pipeline, sample_X):
        """Modelo no debe predecir siempre la misma clase."""
        predictions = trained_pipeline.predict(sample_X)
        unique_predictions = np.unique(predictions)

        assert len(unique_predictions) > 1, \
            "Modelo predice siempre la misma clase"

    def test_model_handles_unseen_categories(self, trained_pipeline):
        """Modelo debe manejar categorías no vistas."""
        # Crear data con categoría nueva
        new_data = pd.DataFrame({
            'CreditScore': [650],
            'Age': [35],
            'Tenure': [5],
            'Balance': [50000.0],
            'NumOfProducts': [2],
            'HasCrCard': [1],
            'IsActiveMember': [1],
            'EstimatedSalary': [75000.0],
            'Geography': ['Italy'],  # Categoría no vista en training
            'Gender': ['Male'],
        })

        # No debe fallar, debe usar handle_unknown='ignore'
        prediction = trained_pipeline.predict(new_data)
        assert len(prediction) == 1

    def test_model_is_deterministic(self, trained_pipeline, sample_X):
        """Predicciones deben ser determinísticas."""
        pred1 = trained_pipeline.predict(sample_X)
        pred2 = trained_pipeline.predict(sample_X)
        np.testing.assert_array_equal(pred1, pred2)


class TestModelFairness:
    """Tests de fairness del modelo."""

    def test_similar_performance_across_genders(self, trained_pipeline, sample_data):
        """Performance similar entre géneros."""
        X = sample_data.drop(columns=['Exited'])
        y = sample_data['Exited']

        male_mask = sample_data['Gender'] == 'Male'
        female_mask = sample_data['Gender'] == 'Female'

        # AUC por género
        if male_mask.sum() > 10 and female_mask.sum() > 10:
            y_proba_male = trained_pipeline.predict_proba(X[male_mask])[:, 1]
            y_proba_female = trained_pipeline.predict_proba(X[female_mask])[:, 1]

            auc_male = roc_auc_score(y[male_mask], y_proba_male)
            auc_female = roc_auc_score(y[female_mask], y_proba_female)

            # Diferencia no debe ser muy grande
            MAX_AUC_DIFF = 0.15
            auc_diff = abs(auc_male - auc_female)

            assert auc_diff <= MAX_AUC_DIFF, \
                f"Diferencia de AUC entre géneros: {auc_diff:.4f}"
```

8.4 Integration Tests: Pipeline Completo

```python
# tests/integration/test_pipeline.py
import pytest
import pandas as pd
import numpy as np
import tempfile
from pathlib import Path
import joblib

class TestPipelineIntegration:
    """Tests de integración del pipeline completo."""

    def test_full_pipeline_train_predict(self, sample_X, sample_y, config):
        """Pipeline completo: entrenar y predecir."""
        from bankchurn.pipeline import build_pipeline

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

        # Train
        pipeline.fit(sample_X, sample_y)

        # Predict
        predictions = pipeline.predict(sample_X)
        probabilities = pipeline.predict_proba(sample_X)

        # Verificaciones
        assert len(predictions) == len(sample_X)
        assert probabilities.shape == (len(sample_X), 2)

    def test_pipeline_serialization(self, trained_pipeline, sample_X):
        """Pipeline debe ser serializable y deserializable."""
        with tempfile.TemporaryDirectory() as tmpdir:
            path = Path(tmpdir) / "pipeline.pkl"

            # Guardar
            joblib.dump(trained_pipeline, path)

            # Cargar
            loaded_pipeline = joblib.load(path)

            # Verificar que funciona igual
            original_preds = trained_pipeline.predict(sample_X)
            loaded_preds = loaded_pipeline.predict(sample_X)

            np.testing.assert_array_equal(original_preds, loaded_preds)

    def test_pipeline_with_missing_values(self, config):
        """Pipeline debe manejar valores faltantes."""
        from bankchurn.pipeline import build_pipeline

        # Data con nulos
        data_with_nulls = pd.DataFrame({
            'CreditScore': [650, None, 700],
            'Age': [35, 40, None],
            'Balance': [50000.0, None, 75000.0],
            'Geography': ['France', 'Germany', None],
            'Gender': ['Male', None, 'Female'],
            'HasCrCard': [1, 0, 1],
        })
        y = pd.Series([0, 1, 0])

        pipeline = build_pipeline(
            numerical_features=['CreditScore', 'Age', 'Balance'],
            categorical_features=['Geography', 'Gender'],
            binary_features=['HasCrCard'],
        )

        # No debe fallar
        pipeline.fit(data_with_nulls, y)
        predictions = pipeline.predict(data_with_nulls)

        assert len(predictions) == 3


class TestTrainerIntegration:
    """Tests de integración del trainer."""

    def test_trainer_produces_model_artifact(self, sample_data, config, tmp_path):
        """Trainer debe producir artefacto de modelo."""
        from bankchurn.training import ChurnTrainer
        from sklearn.model_selection import train_test_split

        X = sample_data.drop(columns=['Exited'])
        y = sample_data['Exited']
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

        config.paths.model_output = tmp_path / "model.pkl"

        trainer = ChurnTrainer(config)
        metrics = trainer.run(X_train, y_train, X_test, y_test)

        # Verificar artefacto
        assert config.paths.model_output.exists()

        # Verificar métricas
        assert 'auc_roc' in metrics
        assert 0 <= metrics['auc_roc'] <= 1
```

8.5 pytest.ini y Configuración

```ini
# pytest.ini
[pytest]
testpaths = tests
python_files = test_*.py
python_classes = Test*
python_functions = test_*
addopts =
    -v
    --tb=short
    --cov=src/bankchurn
    --cov-report=term-missing
    --cov-report=html:reports/coverage
    --cov-fail-under=80
markers =
    slow: marks tests as slow (deselect with '-m "not slow"')
    integration: marks tests as integration tests
    data: marks tests as data quality tests
filterwarnings =
    ignore::DeprecationWarning
```

Ejecutar Tests

```bash
# Todos los tests
pytest

# Solo unit tests (rápidos)
pytest tests/unit/

# Solo integration tests
pytest tests/integration/ -m integration

# Con coverage report
pytest --cov=src/bankchurn --cov-report=html

# Excluir tests lentos
pytest -m "not slow"

# Tests en paralelo
pytest -n auto

# Verbose con detalles de fallos
pytest -v --tb=long
```

8.6 Ejercicio Integrador

Crea Suite de Tests

1. **Unit tests** para tu config y transformers
2. **Data tests** para validar schema
3. **Model tests** con umbrales mínimos
4. **Integration tests** para pipeline completo

Checklist

```
UNIT TESTS:
[ ] Config validation tests
[ ] Transformer tests (fit/transform)
[ ] Utils functions tests
[ ] Coverage > 80%

DATA TESTS:
[ ] Schema validation
[ ] Null checks
[ ] Range validations
[ ] Category validations

MODEL TESTS:
[ ] Prediction shape
[ ] Probability range
[ ] Minimum AUC threshold
[ ] Handles unseen categories

INTEGRATION:
[ ] Full pipeline train/predict
[ ] Serialization works
[ ] Missing values handled
```

Siguiente Paso

Con tests sólidos, es hora de **automatizar con CI/CD**.

Ir a Módulo 09: GitHub Actions Avanzado →

Módulo 08 completado. Tu código ahora tiene red de seguridad.