Detección de Fraude por medio de transacciones financieras

Carolina Salas Moreno , Deykel Bernard Salazar, Esteban Ramírez Montero y Kristhel Porras Mata

Introducción

Sistema para detectar fraudes en transacciones financieras usando un dataset sintético que simula comportamientos reales.

El análisis permite aplicar técnicas de aprendizaje automático para identificar fraudes con precisión y rapidez.

Dataset

- Tamaño: 5,000,000 registros
- Atributos: 18 columnas por registro
- Formato: CSV (~800 MB)
- Datos de transacción: ID, fecha, cuentas, monto y tipo.
- Información de comportamiento y metadatos (ubicación, dispositivo, IP).
- Etiquetas de fraude (binarias y tipos), is_fraud como target.
- Disponible en Kaggle

LEAD UNIVERSITY

Desarrollo del proyecto

Tecnologías Utilizadas

- Apache Spark → Procesamiento rápido y distribuido de grandes volúmenes de datos.
- Lenguaje principal: Python
- Bibliotecas clave: Pandas, Scikit-learn, Matplotlib, PySpark
- Delta Lake → Almacenamiento estructurado, confiable y optimizado para consultas eficientes.

Plataforma de Desarrollo

- Databricks (en la nube) - Free edition

Carga de datos

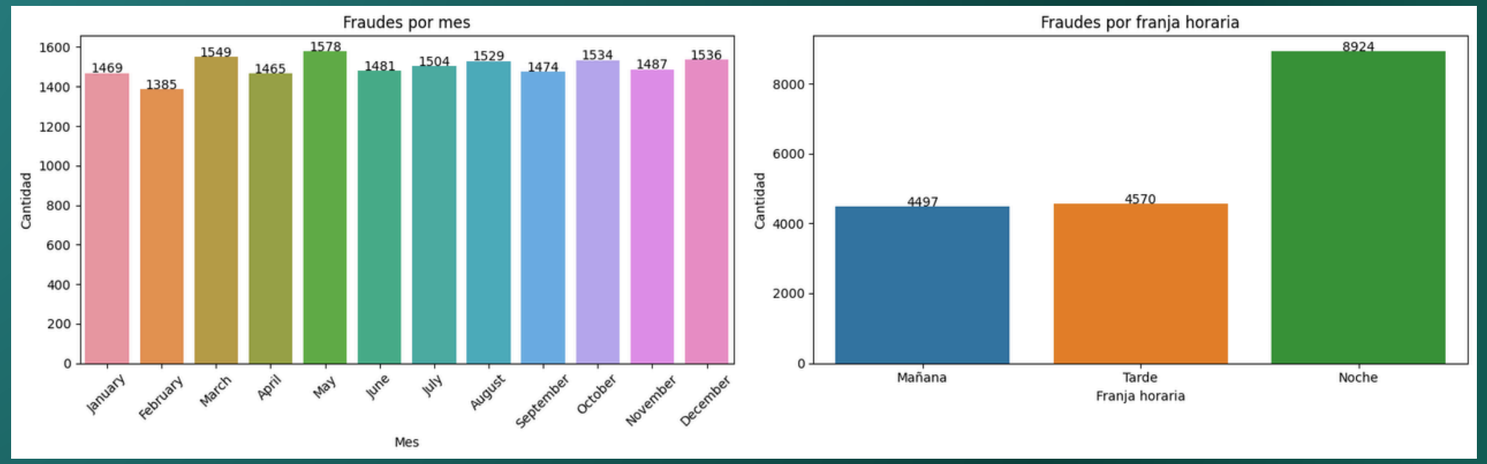
```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import col
spark = SparkSession.builder.getOrCreate()

# Leer datos desde el volumen en Databricks
path = "/Volumes/fraud_project/fraud_detection/fraud_data/"
df_spark = spark.read.option("header", True).csv(path)
df_spark.printSchema()

# Convertir a Pandas solo si es necesario para visualización
for field in df_spark.schema.fields:
    if field.dataType.simpleString() == 'string':
        df_spark = df_spark.withColumn(field.name, col(field.name).cast("string"))
df = df_spark.toPandas()
```

EDA

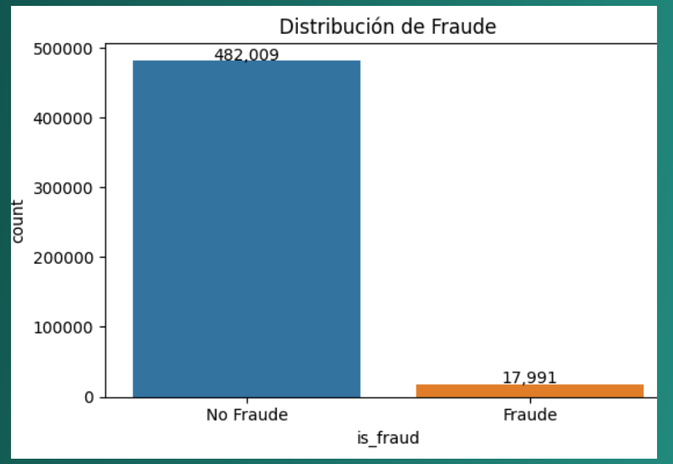
```python
# Exploración básica
print("✅ Dimensiones del dataset:", df.shape)
print("\n📄 Primeros registros:")
display(df.head())

print("\n📄 Tipos de datos:")
print(df.dtypes)

print("\n🔍 Estadísticas numéricas:")
display(df.describe())

print("\n🔍 Estadísticas categóricas:")
#display(df.select_dtypes(include='object').describe())
print(df.select_dtypes(include='object').describe())

# Nulos

print("\n❓ Valores nulos por columna:")
null_counts = df.isnull().sum()
null_percentages = (null_counts / len(df)) * 100
null_summary = pd.DataFrame({
    "Nulos": null_counts,
    "% del total": null_percentages.round(2)
})
null_summary = null_summary.sort_values(by="Nulos", ascending=False)
print(null_summary)

print("\n❓ Filas duplicadas:", df.duplicated().sum())

msno.matrix(df, figsize=(15, 4))
plt.title("Mapa de valores nulos")
plt.show()
```

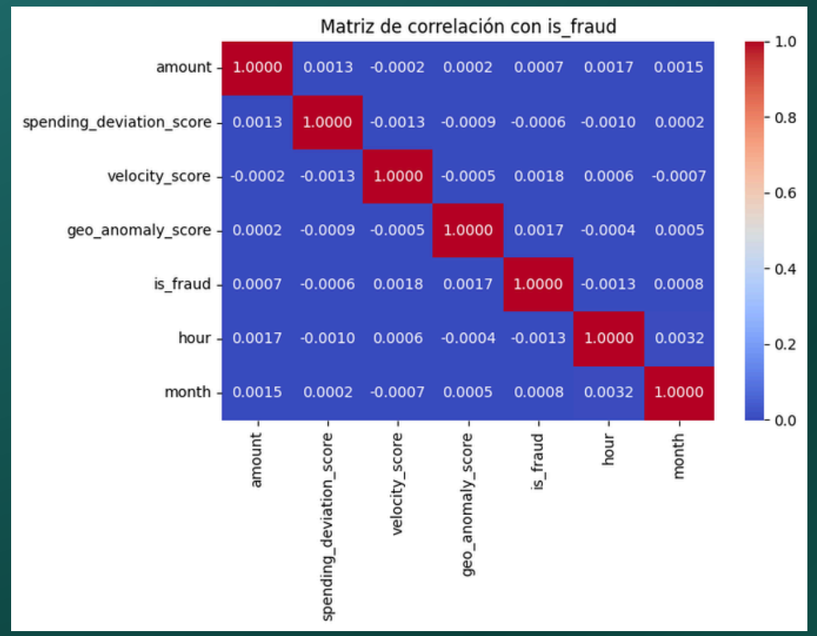
```python
# Conversión de fechas
df['timestamp'] = pd.to_datetime(df['timestamp'], errors='coerce')

# Nueva columna: día, hora, día de la semana, mes y nombre del mes
df['date'] = df['timestamp'].dt.date
df['hour'] = df['timestamp'].dt.hour
df['day_of_week'] = df['timestamp'].dt.day_name()
df['month'] = df['timestamp'].dt.month
df['month_name'] = df['timestamp'].dt.month_name()

# Categorizar horas en franjas horarias
def categorize_time(hour):
    if 0 <= hour < 6:
        return 'Noche'
    elif 6 <= hour < 12:
        return 'Mañana'
    elif 12 <= hour < 18:
        return 'Tarde'
    else:
        return 'Noche'

df['time_period'] = df['hour'].apply(categorize_time)

# Orden de los meses
month_order = ['January', 'February', 'March', 'April', 'May', 'June',
               'July', 'August', 'September', 'October', 'November', 'December']
time_order = ['Mañana', 'Tarde', 'Noche']

# Solo fraudes
fraudes = df[df['is_fraud'] == True]
fig, axes = plt.subplots(1, 2, figsize=(16, 5))

# Eliminar columnas que no agregan valor
df.drop(["fraud_type","time_since_last_transaction"], axis=1, inplace=True)
```

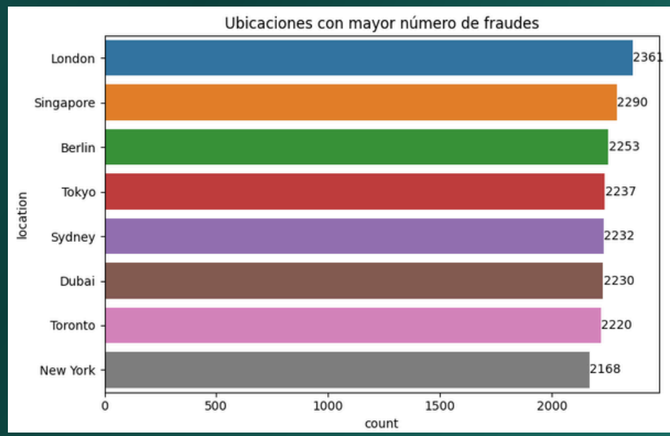
EDA

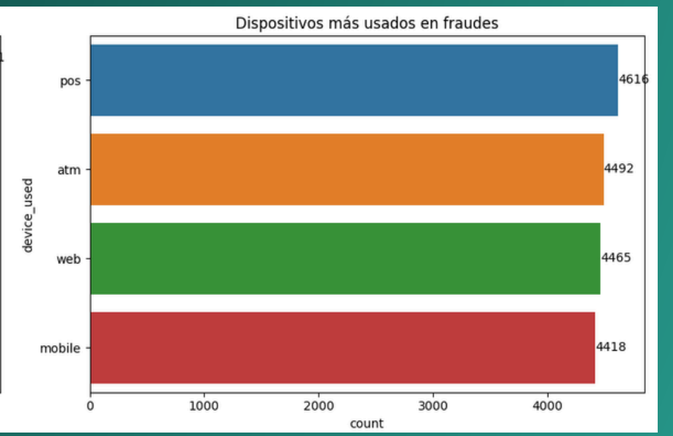

Fraudes por mes
Cantidad
Mes
January
February
March
April
May
June
July
August
September
October
November
December
1469
1385
1549
1465
1578
1481
1504
1529
1474
1534
1487
1536

Fraudes por franja horaria
Cantidad
Franja horaria
Mañana
Tarde
Noche
4497
4570
8924


Distribución de Fraude
count
is_fraud
No Fraude
Fraude
482,009
17,991


Matriz de correlación con is_fraud
amount
spending_deviation_score
velocity_score
geo_anomaly_score
is_fraud
hour
month


Ubicaciones con mayor número de fraudes
location
count
London
Singapore
Berlin
Tokyo
Sydney
Dubai
Toronto
New York
2361
2290
2253
2237
2232
2230
2220
2168


Dispositivos más usados en fraudes
device_used
count
pos
atm
web
mobile
4616
4492
4465
4418

Pre procesamiento

- Conversión y limpieza básica de datos (fechas y tipos numéricos).
- Creación de flags temporales: fin de semana y feriados.
- Codificación cíclica para hora y mes.
- Indicadores clave.
- Cálculo de tiempos entre transacciones por cuenta remitente.
- Estadísticas agregadas por cuenta (promedio, desviación, conteo).
- One-hot encoding (OHE) en variables categóricas.
- Escalado de variables numéricas.
- Eliminación de columnas irrelevantes.
- División y balanceo del dataset.

LEAD UNIVERSITY

PRE PROCESAMIENTO

```python
def preprocesamiento_fraude_big_data(
    df: pd.DataFrame,
    *,
    target_col: str = "fraud_type",
    max_low_card: int = 20,  # Reducido para eficiencia
    large_tx_percentile: float = 0.95,
    test_size: float = 0.20,
    random_state: int = 42,
    balance_strategy: str = "undersample",  # Mejor para datasets grandes
    sample_size: int = None,  # Para sampling inicial si es necesario
    optimize_memory: bool = True,
    verbose: bool = True
):
    if verbose:
        print(f" Procesando dataset con {len(df):,} registros")
        start_time = time.time()

    # === SAMPLING PARA DESARROLLO (OPCIONAL) ===
    if sample_size and len(df) > sample_size:
        if verbose:
            print(f" Tomando muestra estratificada de {sample_size:,} registros para desarrollo...")

        y_full = df[target_col].astype(int)
        df = df.groupby(target_col, group_keys=False).apply(
            lambda x: x.sample(min(len(x), sample_size//2), random_state=random_state)
        ).reset_index(drop=True)

        if verbose:
            print(f"✓ Muestra tomada: {len(df):,} registros")

    # === OPTIMIZACIONES DE MEMORIA ===
    if optimize_memory:
        # Convertir object a category cuando sea eficiente
        for col in df.select_dtypes(include=['object']).columns:
            if col != target_col and df[col].nunique() / len(df) < 0.5:
                df[col] = df[col].astype('category')

    def canon(s: str) -> str:
        s = unicodedata.normalize("NFKC", str(s)).strip().lower()
        return re.sub(r"\s+", " ", s)

    label_rx = re.compile(r"(fraud|label|target|outcome|chargeback)", re.I)
    df = df.copy()
```

```python
# === LIMPIEZA NUMÉRICA EFICIENTE ===
def to_float_vectorized(series):
    """Versión vectorizada más eficiente"""
    if series.dtype in ['int64', 'float64']:
        return series

    # Convertir a string y limpiar
    str_series = series.astype(str).str.replace(',', '').str.replace('$', '')
    # Usar pd.to_numeric que es más eficiente
    return pd.to_numeric(str_series, errors='coerce')

numeric_cols = ["amount", "time_since_last_transaction",
                "spending_deviation_score", "velocity_score", "geo_anomaly_score"]

for c in numeric_cols:
    if c in df.columns:
        if verbose and len(df) > 100000:
            print(f" Procesando {c}...")
        df[c] = to_float_vectorized(df[c])

# === TARGET PROCESSING ===
if target_col not in df.columns:
    raise ValueError(f"target_col '{target_col}' no está en df.columns")

if df[target_col].dtype in ("object", "category"):
    s = df[target_col].astype(str).str.strip().str.lower()
    negatives = {"", "nan", "none", "null", "0", "false", "no", "normal"}
    df[target_col] = (~s.isin(negatives)).astype(int)
else:
    df[target_col] = pd.to_numeric(df[target_col], errors="coerce").fillna(0).astype(int)
```

PRE PROCESAMIENTO

```python
# === CARACTERÍSTICAS TEMPORALES EFICIENTES ===
if "timestamp" in df.columns:
    if verbose:
        print(" Creando características temporales...")

    df["timestamp"] = pd.to_datetime(df["timestamp"], errors="coerce")

    # Solo las características más importantes para eficiencia
    df["hour"] = df["timestamp"].dt.hour
    df["day_of_week"] = df["timestamp"].dt.dayofweek

    # Características binarias eficientes
    df["is_business_hour"] = ((df["hour"] >= 9) & (df["hour"] <= 17)).astype(np.int8)
    df["is_weekend"] = (df["day_of_week"] >= 5).astype(np.int8)
    df["is_late_night"] = ((df["hour"] >= 22) | (df["hour"] <= 6)).astype(np.int8)

    # Solo encoding cíclico esencial
    df["hour_sin"] = np.sin(2 * np.pi * df["hour"] / 24)
    df["hour_cos"] = np.cos(2 * np.pi * df["hour"] / 24)
```

```python
# === SPLIT ESTRATIFICADO TEMPRANO ===
if verbose:
    print(" Realizando split estratificado...")

y_full = df[target_col].astype(int)
idx_train, idx_test = train_test_split(
    df.index, stratify=y_full, test_size=test_size, random_state=random_state
)
df_train = df.loc[idx_train].copy()
df_test = df.loc[idx_test].copy()

if verbose:
    fraud_rate = y_full.mean()
    print(f"✓ Train: {len(df_train):,} | Test: {len(df_test):,}")
    print(f"✓ Tasa de fraude: {fraud_rate:.1%}")
```

```python
# === FEATURE ENGINEERING EFICIENTE (SOLO LO ESENCIAL) ===
if {"sender_account", "amount"}.issubset(df.columns):
    if verbose:
        print(" Creando estadísticas por cuenta...")

    # Solo estadísticas esenciales para eficiencia
    stats = df_train.groupby("sender_account")["amount"].agg([
        "mean", "count", "std"
    ])
    stats.columns = ["sender_avg_amount", "sender_tx_count", "sender_std_amount"]

    # Feature derivada simple
    stats["sender_amount_cv"] = stats["sender_std_amount"] / (stats["sender_avg_amount"] + 1e-6)

    df_train = df_train.join(stats, on="sender_account")
    df_test = df_test.join(stats, on="sender_account")

    # Z-score de la transacción actual
    df_train["amount_zscore"] = (df_train["amount"] - df_train["sender_avg_amount"]) / (df_train["sender_std_amount"] + 1e-6)
    df_test["amount_zscore"] = (df_test["amount"] - df_test["sender_avg_amount"]) / (df_test["sender_std_amount"] + 1e-6)

# Características de dispositivo/ubicación simplificadas
if "device_used" in df.columns and "sender_account" in df.columns:
    dev_counts = df_train.groupby("sender_account")["device_used"].nunique()
    df_train["device_diversity"] = df_train["sender_account"].map(dev_counts).fillna(1)
    df_test["device_diversity"] = df_test["sender_account"].map(dev_counts).fillna(1)
    df_train["multiple_devices"] = (df_train["device_diversity"] > 1).astype(np.int8)
    df_test["multiple_devices"] = (df_test["device_diversity"] > 1).astype(np.int8)
```

```python
# === PREPARACIÓN DE CARACTERÍSTICAS ===
base_exclude = {
    target_col, "is_fraud", "transaction_id", "sender_account", "receiver_account",
    "ip_address", "device_hash", "timestamp", "device_used", "location"
}

label_like = {c for c in df.columns if label_rx.search(canon(c))}
label_like.discard(target_col)
no_modelar = (base_exclude | label_like) & set(df.columns)

drop_cols = [c for c in no_modelar if c in df.columns]
X_train = df_train.drop(columns=drop_cols, errors="ignore")
X_test = df_test.drop(columns=drop_cols, errors="ignore")
y_train = df_train[target_col].astype(int)
y_test = df_test[target_col].astype(int)
```

PRE PROCESAMIENTO

```python
# === PROCESAMIENTO CATEGÓRICO EFICIENTE ===
if verbose:
    print(" Procesando variables categóricas...")

all_cats = [c for c in X_train.columns if X_train[c].dtype in ("object", "category")]
cat_low = [c for c in all_cats if X_train[c].nunique(dropna=True) <= max_low_card]
cat_high = [c for c in all_cats if c not in cat_low]

# Filtrar Label-like
cat_low = [c for c in cat_low if not label_rx.search(canon(c))]
cat_high = [c for c in cat_high if not label_rx.search(canon(c))]

base_num_cols = [c for c in X_train.columns if c not in all_cats]

# Imputación eficiente
for c in (cat_low + cat_high):
    X_train[c] = X_train[c].fillna("Unknown")
    X_test[c] = X_test[c].fillna("Unknown")

for c in base_num_cols:
    if X_train[c].dtype in ['object', 'category']:
        continue
    median_val = X_train[c].median()
    X_train[c] = X_train[c].fillna(median_val)
    X_test[c] = X_test[c].fillna(median_val)

# Umbral de transacción grande
large_tx_threshold = None
if "amount" in X_train.columns:
    large_tx_threshold = X_train["amount"].quantile(large_tx_percentile)
    X_train["is_large_transaction"] = (X_train["amount"] > large_tx_threshold).astype(np.int8)
    X_test["is_large_transaction"] = (X_test["amount"] > large_tx_threshold).astype(np.int8)
    base_num_cols.append("is_large_transaction")
```

```python
# === ENCODING EFICIENTE ===
artefactos = {}

# One-Hot Encoding solo para baja cardinalidad
ohe = None
trn_oh = tst_oh = None
if cat_low:
    if verbose:
        print(f" One-hot encoding para {len(cat_low)} columnas...")

    try:
        ohe = OneHotEncoder(handle_unknown="ignore", drop="first", sparse_output=True, dtype=np.int8)
    except TypeError:
        ohe = OneHotEncoder(handle_unknown="ignore", drop="first", sparse=True, dtype=np.int8)

    ohe.fit(X_train[cat_low])
    trn_oh = ohe.transform(X_train[cat_low])
    tst_oh = ohe.transform(X_test[cat_low])
    artefactos["ohe"] = ohe
    artefactos["ohe_cols"] = list(cat_low)

# Frequency encoding para alta cardinalidad (más eficiente que target encoding)
fe_maps = {}
if cat_high:
    if verbose:
        print(f" Frequency encoding para {len(cat_high)} columnas...")

    for c in cat_high:
        freq = X_train[c].value_counts(normalize=True)
        fe_maps[c] = freq.to_dict()
        X_train[c + "_freq"] = X_train[c].map(fe_maps[c]).fillna(0.0)
        X_test[c + "_freq"] = X_test[c].map(fe_maps[c]).fillna(0.0)

artefactos["freq_encoding_maps"] = fe_maps

# Eliminar categóricas originales
X_train_drop = X_train.drop(columns=cat_low + cat_high, errors="ignore")
X_test_drop = X_test.drop(columns=cat_low + cat_high, errors="ignore")
```

PRE PROCESAMIENTO

```python
# === ESCALADO EFICIENTE ===
if verbose:
    print(" Escalando características numéricas...")

scaler = RobustScaler()
scale_cols = [c for c in base_num_cols if c in X_train_drop.columns]

if scale_cols:
    numeric_mask = X_train_drop[scale_cols].select_dtypes(include=[np.number]).columns
    scale_cols = [c for c in scale_cols if c in numeric_mask]

    if scale_cols:
        scaler.fit(X_train_drop[scale_cols])
        X_train_drop.loc[:, scale_cols] = scaler.transform(X_train_drop[scale_cols])
        X_test_drop.loc[:, scale_cols] = scaler.transform(X_test_drop[scale_cols])

artefactos["scaler"] = scaler
artefactos["scale_cols"] = scale_cols

# Limpiar NaN finales
X_train_drop = X_train_drop.fillna(0)
X_test_drop = X_test_drop.fillna(0)

# === CREAR MATRICES SPARSE ===
X_train_dense = X_train_drop.copy()
X_test_dense = X_test_drop.copy()

X_train_num = sp.csr_matrix(X_train_drop.values, dtype=np.float32)  # float32 para memoria
X_test_num = sp.csr_matrix(X_test_drop.values, dtype=np.float32)

if trn_oh is not None:
    X_train_sparse = sp.hstack([X_train_num, trn_oh], format="csr")
    X_test_sparse = sp.hstack([X_test_num, tst_oh], format="csr")
else:
    X_train_sparse, X_test_sparse = X_train_num, X_test_num
```

```python
# === BALANCEO OPTIMIZADO PARA BIG DATA ===
if verbose:
    print(f" Aplicando estrategia de balanceo: {balance_strategy}")

y_train_res = y_train.copy()
X_train_sparse_res = X_train_sparse
X_train_dense_res = X_train_dense
sample_weight = None

if balance_strategy == "undersample":
    # Undersample inteligente preservando diversidad
    rus = RandomUnderSampler(
        random_state=random_state,
        sampling_strategy=0.3  # Ratio más conservador para preservar información
    )
    idx = np.arange(len(y_train_res)).reshape(-1, 1)
    idx_res, y_train_res = rus.fit_resample(idx, y_train_res)
    sel = idx_res.ravel()
    X_train_sparse_res = X_train_sparse[sel]
    X_train_dense_res = X_train_dense.iloc[sel]

    if verbose:
        print(f" Undersample: {len(y_train_res):,} registros finales")

elif balance_strategy == "weights":
    sample_weight = compute_sample_weight(class_weight="balanced", y=y_train_res)
    if verbose:
        print(" Class weights aplicados")

else:
    if verbose:
        print(" Sin balanceo aplicado")
```

PRE PROCESAMIENTO

```python
# === ARTEFACTOS FINALES ===
artefactos.update({
    "large_tx_threshold": large_tx_threshold,
    "base_num_cols": base_num_cols,
    "cat_low": list(cat_low),
    "cat_high": list(cat_high),
    "drop_cols": drop_cols,
    "sample_weight": sample_weight,
    "balance_strategy": balance_strategy,
    "X_train_dense": X_train_dense,
    "X_test_dense": X_test_dense,
    "X_train_dense_res": X_train_dense_res,
})

if verbose:
    elapsed = time.time() - start_time
    print(f" Preprocesamiento completado en {elapsed:.1f} segundos")
    print(f" Shape final train: {X_train_sparse_res.shape}")
    print(f" Shape final test: {X_test_sparse.shape}")

return X_train_sparse_res, X_test_sparse, y_train_res, y_test, artefactos
```

Optimización

- Se identificó que usar el umbral por defecto (0.5) en detección de fraude genera demasiados falsos positivos, baja precisión (3–5%) y sobrecarga operativa.
- Se diseñaron estrategias de optimización de umbrales con distintos enfoques: precision target, recall target, max F1, conservative y balanced.
- Se estableció un proceso de selección jerárquico, priorizando la precisión (precision target → conservative → max F1 → balanced) debido al alto costo de los falsos positivos frente al ahorro de los verdaderos positivos.
- Se definieron métricas de negocio clave: número total de alertas, porcentaje de precisión y estatus de cumplimiento de objetivos.
- Se modeló el impacto esperado al aplicar umbrales optimizados:
  - Precisión: aumento de 3–5% → 15–30%
  - Reducción de alertas: 70–90% menos
  - Mejora en F1-score
  - Cambio en ROI: de negativo a positivo

LEAD UNIVERSITY

Optimización

```python
def optimizar_precision_fraude(modelos, X_test_sparse, y_test, X_test_dense=None,
                               target_precision=0.20, target_recall=0.60):

    print(" OPTIMIZACIÓN DE PRECISIÓN Y UMBRAL")
    print("=" * 60)
    print(f" Target Precision: {target_precision:.1%}")
    print(f" Target Recall: {target_recall:.1%}")
    print()

    resultados_optimizados = []
    mejores_umbrales = {}

    for nombre, modelo in modelos.items():
        print(f" Optimizando {nombre}...")

        # Seleccionar datos apropiados
        if "Gradient Boosting" in nombre and X_test_dense is not None:
            X_test_model = X_test_dense
        else:
            X_test_model = X_test_sparse

        # Obtener probabilidades
        if hasattr(modelo, 'predict_proba'):
            y_probs = modelo.predict_proba(X_test_model)[:, 1]
        else:
            print(f"     {nombre} no tiene predict_proba, usando decisión por defecto")
            y_pred_default = modelo.predict(X_test_model)
            precision_default = precision_score(y_test, y_pred_default, zero_division=0)
            recall_default = recall_score(y_test, y_pred_default, zero_division=0)
            f1_default = f1_score(y_test, y_pred_default, zero_division=0)

            resultados_optimizados.append({
                'Modelo': nombre,
                'Umbral': 'default',
                'Precision': f"{precision_default:.4f}",
                'Recall': f"{recall_default:.4f}",
                'F1-Score': f"{f1_default:.4f}",
                'Status': 'No optimizable'
            })
            continue

        # Calcular curva precision-recall
        precisions, recalls, thresholds = precision_recall_curve(y_test, y_probs)
        # Encontrar mejores umbrales según diferentes criterios
        umbrales_candidatos = {}
```

```python
# 1. Umbral para precision target
mask_precision = precisions >= target_precision
if mask_precision.any():
    idx_precision = np.where(mask_precision)[0]
    # Entre los que cumplen precision, elegir el de mayor recall
    best_idx = idx_precision[np.argmax(recalls[idx_precision])]
    umbrales_candidatos['precision_target'] = {
        'threshold': thresholds[best_idx] if best_idx < len(thresholds) else 0.5,
        'precision': precisions[best_idx],
        'recall': recalls[best_idx],
        'f1': 2 * precisions[best_idx] * recalls[best_idx] / (precisions[best_idx] + recalls[best_idx])
    }

# 2. Umbral para recall target
mask_recall = recalls >= target_recall
if mask_recall.any():
    idx_recall = np.where(mask_recall)[0]
    # Entre los que cumplen recall, elegir el de mayor precision
    best_idx = idx_recall[np.argmax(precisions[idx_recall])]
    umbrales_candidatos['recall_target'] = {
        'threshold': thresholds[best_idx] if best_idx < len(thresholds) else 0.5,
        'precision': precisions[best_idx],
        'recall': recalls[best_idx],
        'f1': 2 * precisions[best_idx] * recalls[best_idx] / (precisions[best_idx] + recalls[best_idx])
    }

# 3. Umbral que maximiza F1
f1_scores = 2 * precisions * recalls / (precisions + recalls + 1e-8)
best_f1_idx = np.argmax(f1_scores)
umbrales_candidatos['max_f1'] = {
    'threshold': thresholds[best_f1_idx] if best_f1_idx < len(thresholds) else 0.5,
    'precision': precisions[best_f1_idx],
    'recall': recalls[best_f1_idx],
    'f1': f1_scores[best_f1_idx]
}

# 4. Umbral balanceado (precision ~= recall)
balance_diff = np.abs(precisions - recalls)
balanced_idx = np.argmin(balance_diff)
umbrales_candidatos['balanced'] = {
    'threshold': thresholds[balanced_idx] if balanced_idx < len(thresholds) else 0.5,
    'precision': precisions[balanced_idx],
    'recall': recalls[balanced_idx],
    'f1': f1_scores[balanced_idx]
}
```

Optimización

```python
# 5. Umbral conservador (alta precision, recall moderado)
        conservative_mask = precisions >= 0.15  # Al menos 15% precision
        if conservative_mask.any():
            conservative_idx = np.where(conservative_mask)[0]
            best_idx = conservative_idx[np.argmax(recalls[conservative_idx])]
            umbrales_candidatos['conservative'] = {
                'threshold': thresholds[best_idx] if best_idx < len(thresholds) else 0.7,
                'precision': precisions[best_idx],
                'recall': recalls[best_idx],
                'f1': f1_scores[best_idx]
            }

        # Seleccionar el mejor umbral (priorizar precision target si existe)
        if 'precision_target' in umbrales_candidatos:
            mejor_umbral = umbrales_candidatos['precision_target']
            criterio = 'precision_target'
        elif 'conservative' in umbrales_candidatos:
            mejor_umbral = umbrales_candidatos['conservative']
            criterio = 'conservative'
        elif 'max_f1' in umbrales_candidatos:
            mejor_umbral = umbrales_candidatos['max_f1']
            criterio = 'max_f1'
        else:
            mejor_umbral = umbrales_candidatos['balanced']
            criterio = 'balanced'

        mejores_umbrales[nombre] = mejor_umbral['threshold']

        # Evaluar con el mejor umbral
        y_pred_opt = (y_probs >= mejor_umbral['threshold']).astype(int)

        # Métricas finales
        precision_final = precision_score(y_test, y_pred_opt, zero_division=0)
        recall_final = recall_score(y_test, y_pred_opt, zero_division=0)
        f1_final = f1_score(y_test, y_pred_opt, zero_division=0)
        roc_auc = roc_auc_score(y_test, y_probs)

        # Conteos
        tp = ((y_pred_opt == 1) & (y_test == 1)).sum()
        fp = ((y_pred_opt == 1) & (y_test == 0)).sum()
        fn = ((y_pred_opt == 0) & (y_test == 1)).sum()
        tn = ((y_pred_opt == 0) & (y_test == 0)).sum()
```

```python
        # Métricas de negocio
        precision_pct = precision_final * 100
        total_alerts = tp + fp
        fraud_detection_rate = tp / (tp + fn) if (tp + fn) > 0 else 0

        status = " Cumple targets" if precision_final >= target_precision and recall_final >= target_recall else \
                 " Precision baja" if precision_final < target_precision else \
                 " Recall bajo" if recall_final < target_recall else " No cumple"

        resultados_optimizados.append({
            'Modelo': nombre,
            'Umbral': f"{mejor_umbral['threshold']:.3f}",
            'Criterio': criterio,
            'Precision': f"{precision_final:.4f}",
            'Recall': f"{recall_final:.4f}",
            'F1-Score': f"{f1_final:.4f}",
            'ROC-AUC': f"{roc_auc:.4f}",
            'TP': tp,
            'FP': fp,
            'FN': fn,
            'Total_Alerts': total_alerts,
            'Precision_%': f"{precision_pct:.1f}%",
            'Status': status
        })

        print(f"    Mejor umbral: {mejor_umbral['threshold']:.3f} ({criterio})")
        print(f"    Precision: {precision_final:.1%} | Recall: {recall_final:.1%} | F1: {f1_final:.3f}")
        print(f"    Alertas totales: {total_alerts:,} (TP: {tp:,}, FP: {fp:,})")
        print()

    return pd.DataFrame(resultados_optimizados), mejores_umbrales
```

Modelos y métricas

Modelos utilizados

- Logistic Regression
- Random Forest
- Gradient Boosting

Métricas utilizadas

- Precisión
- Sensibilidad
- F1-Score

LEAD UNIVERSITY

Entrenamiento

```python
def entrenar_con_class_weights(X_train_sparse, y_train, X_train_dense, sample_weight=None):
    print(" RE-ENTRENANDO CON CLASS WEIGHTS (sin undersample)...")
    print("=" * 60)

    from sklearn.linear_model import LogisticRegression
    from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier

    modelos_weights = {}

    # Logistic Regression con class weight
    print(" Logistic Regression con class_weight='balanced'...")
    lr_balanced = LogisticRegression(
        solver="saga",
        penalty="l2",
        C=0.01,  # Más regularización para reducir overfitting
        max_iter=500,
        tol=1e-4,
        class_weight='balanced',  # ← Clave para reducir FP
        random_state=42,
        n_jobs=-1
    )
    modelos_weights["Logistic Regression Balanced"] = lr_balanced.fit(X_train_sparse, y_train)

    # Random Forest con class weight más conservador
    print(" Random Forest con class_weight='balanced_subsample'...")
    rf_balanced = RandomForestClassifier(
        n_estimators=300,
        max_depth=12,  # Más conservador
        min_samples_split=50,  # Más restrictivo
        min_samples_leaf=20,   # Más restrictivo
        max_features="sqrt",
        bootstrap=True,
        max_samples=0.6,  # Menos muestras por árbol
        class_weight='balanced_subsample',
        n_jobs=-1,
        random_state=42
    )
    modelos_weights["Random Forest Balanced"] = rf_balanced.fit(X_train_sparse, y_train)

    # Gradient Boosting más conservador
    if X_train_dense is not None:
        print(" Gradient Boosting conservador...")
        gbt_conservative = GradientBoostingClassifier(
            n_estimators=200,
            learning_rate=0.05,  # Menor learning rate
            max_depth=4,         # Menos profundidad
            subsample=0.7,
            max_features="sqrt",
            min_samples_split=100,  # Más restrictivo
            min_samples_leaf=50,    # Más restrictivo
            random_state=42
        )
        modelos_weights["Gradient Boosting Conservative"] = gbt_conservative.fit(X_train_dense, y_train)

    print(f" {len(modelos_weights)} modelos re-entrenados con class weights")
    return modelos_weights
```

```
Optimizando Logistic Regression Balanced...
   Mejor umbral: 0.500 (precision_target)
   Precision: 3.6% | Recall: 33.9% | F1: 0.066
   Alertas totales: 33,401 (TP: 1,218, FP: 32,183)

Optimizando Random Forest Balanced...
   Mejor umbral: 0.500 (precision_target)
   Precision: 4.4% | Recall: 51.9% | F1: 0.080
   Alertas totales: 42,896 (TP: 1,866, FP: 41,030)

Optimizando Gradient Boosting Conservative...
   Mejor umbral: 0.204 (precision_target)
   Precision: 15.4% | Recall: 0.1% | F1: 0.001
   Alertas totales: 13 (TP: 2, FP: 11)
```

Análisis ROI

Conceptos clave

Verdadero Positivo (TP): Fraude detectado correctamente
- Beneficio: Evitamos pérdida por fraude + costos de investigación

Falso Positivo (FP): Transacción legítima marcada como fraude
- Costo: Fricción al cliente + tiempo de revisión + posible pérdida de cliente

Falso Negativo (FN): Fraude no detectado
- Costo implícito: Pérdida total del fraude (no calculado aquí directamente)

Verdadero Negativo (TN): Transacción legítima correctamente clasificada
- Sin costo ni beneficio directo

LEAD UNIVERSITY

Análisis ROI

```python
def analisis_costo_beneficio(resultados_df, costo_fp=100, beneficio_tp=1000):
    """
    Análisis de costo-beneficio para seleccionar el mejor modelo
    """
    print(f"\n ANÁLISIS COSTO-BENEFICIO")
    print("=" * 40)
    print(f"Costo por Falso Positivo: ${costo_fp}")
    print(f"Beneficio por Verdadero Positivo: ${beneficio_tp}")
    print()

    analisis = []

    for _, row in resultados_df.iterrows():
        if row['Status'] == 'No optimizable':
            continue

        tp = int(row['TP'])
        fp = int(row['FP'])

        costo_total = fp * costo_fp
        beneficio_total = tp * beneficio_tp
        beneficio_neto = beneficio_total - costo_total
        roi = (beneficio_neto / costo_total * 100) if costo_total > 0 else 0

        analisis.append({
            'Modelo': row['Modelo'],
            'Beneficio_Total': f"${beneficio_total:,}",
            'Costo_Total': f"${costo_total:,}",
            'Beneficio_Neto': f"${beneficio_neto:,}",
            'ROI_%': f"{roi:.1f}%",
            'Precision_%': row['Precision_%']
        })

    df_analisis = pd.DataFrame(analisis)
    print(df_analisis.to_string(index=False))

    if len(df_analisis) > 0:
        mejor_roi_idx = df_analisis['ROI_%'].str.rstrip('%').astype(float).idxmax()
        mejor_modelo_negocio = df_analisis.iloc[mejor_roi_idx]
        print(f"\n MEJOR MODELO POR ROI: {mejor_modelo_negocio['Modelo']}")
        print(f" ROI: {mejor_modelo_negocio['ROI_%']}")
        print(f" Beneficio Neto: {mejor_modelo_negocio['Beneficio_Neto']}")

    return df_analisis
```

```
Costo por Falso Positivo: $100
Beneficio por Verdadero Positivo: $1000

                       Modelo Beneficio_Total Costo_Total Beneficio_Neto  ROI_% Precision_%
  Logistic Regression Balanced      $1,218,000  $3,218,300    $-2,000,300 -62.2%         3.6%
        Random Forest Balanced      $1,866,000  $4,103,000    $-2,237,000 -54.5%         4.4%
Gradient Boosting Conservative          $2,000      $1,100            $900  81.8%        15.4%

 MEJOR MODELO POR ROI: Gradient Boosting Conservative
 ROI: 81.8%
 Beneficio Neto: $900
```

Flujo completo y resultados

- Pipeline con optimización
- Resultados por fases

Flujo y resultados

```python
def pipeline_optimizacion_completa(df, target_col="fraud_type"):
    """
    Pipeline completo: re-entrenamiento + optimización de umbral
    """

    print(" PIPELINE COMPLETO DE OPTIMIZACIÓN")
    print("=" * 70)

    # 1. Re-preprocesamiento con class weights
    print("\n=== FASE 1: RE-PREPROCESAMIENTO CON CLASS WEIGHTS ===")

    # Importar función de preprocesamiento (asumiendo que ya existe)
    X_train_res, X_test_tr, y_train_res, y_test, artefactos = preprocesamiento_fraude_big_data(
        df,
        target_col=target_col,
        balance_strategy="weights",  # ← Cambio clave
        max_low_card=15,
        optimize_memory=True,
        verbose=True
    )

    # 2. Re-entrenamiento con configuraciones más conservadoras
    print("\n=== FASE 2: RE-ENTRENAMIENTO CONSERVADOR ===")
    modelos_optimizados = entrenar_con_class_weights(
        X_train_sparse=X_train_res,
        y_train=y_train_res,
        X_train_dense=artefactos["X_train_dense_res"]
    )

    # 3. Optimización de umbrales
    print("\n=== FASE 3: OPTIMIZACIÓN DE UMBRALES ===")
    resultados_opt, umbrales = optimizar_precision_fraude(
        modelos_optimizados,
        X_test_sparse=X_test_tr,
        y_test=y_test,
        X_test_dense=artefactos["X_test_dense"],
        target_precision=0.15,  # 15% precision mínima
        target_recall=0.50      # 50% recall mínimo
    )

    # 4. Análisis de negocio
    print("\n=== FASE 4: ANÁLISIS DE NEGOCIO ===")
    analisis_negocio = analisis_costo_beneficio(resultados_opt)

    # 5. Comparación con resultados anteriores
    print("\n=== COMPARACIÓN CON RESULTADOS ANTERIORES ===")
    print(" ANTES (con undersample):")
    print("   Precision: 0.035-0.044 (3.5-4.4%)")
    print("   Recall: 0.63-0.69")
    print("   F1: 0.066-0.089")
    print("   Falsos Positivos: 46K+")
    print()
    print(" AHORA (con class weights + umbral optimizado):")
    print(resultados_opt[['Modelo', 'Precision_%', 'Recall', 'F1-Score', 'Total_Alerts', 'Status']].to_string(index=False))

    return resultados_opt, umbrales, analisis_negocio
```

```
=== COMPARACIÓN CON RESULTADOS ANTERIORES ===
 ANTES (con undersample):
   Precision: 0.035-0.044 (3.5-4.4%)
   Recall: 0.63-0.69
   F1: 0.066-0.089
   Falsos Positivos: 46K+

 AHORA (con class weights + umbral optimizado):
                      Modelo Precision_% Recall F1-Score  Total_Alerts          Status
  Logistic Regression Balanced        3.6% 0.3385   0.0658         33401  Precision baja
       Random Forest Balanced        4.4% 0.5186   0.0803         42896  Precision baja
Gradient Boosting Conservative       15.4% 0.0006   0.0011            13     Recall bajo

 RECOMENDACIÓN FINAL:
1. Usar el modelo con mejor ROI del análisis de negocio
2. Aplicar el umbral optimizado correspondiente
3. Monitorear precision en producción y ajustar si es necesario
```

Desafíos y presentación del código

- Manejo de desbalance de los datos
- Decision entre Series de Panda vs Dataframes de Spark
- Limitaciones de recursos para manejar gran cantidad de datos con la version Free Edition

Código completo...

Ingrese aquí