UNIVERSIDAD TECNOLÓGICA METROPOLITANA
UTEM del Estado de Chile

INFB6052 - HERRAMIENTAS PARA CS. DE DATOS
INGENIERÍA CIVIL EN CIENCIA DE DATOS

Informe Primera Parte - Prueba 1:

Implementación de 5 Ejercicios Prácticos en Ciencia de Datos

Ignacio Ramírez
Cristian Vergara
Antonia Montecinos
Grupo 2

Segundo Semestre de 2025

Informe de la Primera Parte:

(i) Introducción

Este informe presenta la documentación técnica de la **Primera Parte** de la Prueba 1 del curso INFB6052 - Herramientas para Ciencia de Datos. El trabajo consta de cinco ejercicios prácticos que abarcan aspectos fundamentales del ecosistema de herramientas modernas para el análisis y procesamiento de datos a gran escala.

Objetivos Generales

El objetivo principal de esta primera parte es demostrar competencia práctica en:

- **Comparación de paradigmas de almacenamiento**: SQL vs NoSQL en contextos reales.
- **Manejo de grandes volúmenes de datos**: Técnicas de ingestión y procesamiento de datasets superiores a 200 MB sin almacenamiento local completo.
- **Evaluación de frameworks de procesamiento**: Comparación empírica entre Pandas y PySpark.
- **Análisis de librerías de visualización**: Evaluación sistemática de Matplotlib, Seaborn y Plotly.
- **Implementación de algoritmos desde cero**: Desarrollo del Perceptrón sin librerías de Machine Learning.

Estructura del Informe

El documento está organizado en cinco secciones principales, cada una correspondiente a un ejercicio:

1. **Comparación SQL vs NoSQL**: Análisis de bases de datos tabulares y no tabulares.
2. **Pipeline de Ingestión de Datos Grandes**: Estrategias para datasets $\geq$ 200 MB.
3. **Pandas vs PySpark**: Comparación de rendimiento y casos de uso.
4. **Librerías de Visualización**: Evaluación de Matplotlib, Seaborn y Plotly.
5. **Perceptrón desde Cero**: Implementación y validación del algoritmo clásico.

Cada sección incluye: objetivos específicos, metodología, implementación técnica, resultados y conclusiones.

(ii) Ejercicio 1: Comparación de Bases de Datos SQL vs NoSQL

Objetivos

- Comparar el rendimiento de bases de datos tabulares (SQL) y no tabulares (NoSQL) en operaciones comunes de carga y consulta.
- Analizar datasets de diferentes naturalezas: datos estructurados (transacciones financieras) y datos no estructurados (texto).
- Evaluar ventajas y desventajas de cada paradigma en contextos específicos.

Datasets Utilizados

Dataset Tabular: Fraude en Transacciones Bancarias

Fuente: Kaggle - Fraud Detection Dataset
Características:

- **Archivos:** `fraudTrain.csv` y `fraudTest.csv`
- **Total de registros:** Aproximadamente 1,850,000 transacciones combinadas
- **Columnas principales:**
  - `trans_date_trans_time`: Timestamp de la transacción
  - `merchant`: Comerciante
  - `category`: Categoría de compra
  - `amt`: Monto de la transacción
  - `lat`, `long`: Coordenadas geográficas
  - `is_fraud`: Etiqueta binaria (0 = legítima, 1 = fraudulenta)
- **Periodo:** 1 de enero de 2019 - 31 de diciembre de 2020

Justificación: Dataset ideal para demostrar el rendimiento de bases de datos relacionales en operaciones de agregación, filtrado y joins sobre datos estructurados.

Dataset No Tabular: WikiSent2

Fuente: Corpus de texto no estructurado derivado de Wikipedia
Características:

- **Archivo:** `wikisent2.txt`
- **Contenido:** Aproximadamente 500,000 líneas de texto en formato libre
- **Estructura:** Sin esquema fijo, texto puro con variabilidad en longitud y formato

Justificación: Representa el tipo de datos que requiere flexibilidad de esquema, ideal para bases NoSQL orientadas a documentos.

Implementación

Herramientas Utilizadas

- **Pandas:** Carga y análisis exploratorio de datos tabulares
- **SQLite:** Base de datos relacional embebida para comparación SQL
- **MongoDB:** Base de datos NoSQL orientada a documentos (simulado con diccionarios Python para el análisis)
- **Matplotlib:** Visualización de métricas comparativas

Flujo de Trabajo

El notebook `tab-vs-notab.ipynb` implementa el siguiente pipeline:

1. **Carga de Datos Tabulares:**
   - Lectura de `fraudTrain.csv` y `fraudTest.csv` con Pandas
   - Concatenación de ambos datasets con columna identificadora `dataset`
   - Medición de tiempo de carga
2. **Exploración Básica:**
   - Análisis de tipos de datos (`df.info()`)
   - Detección de valores nulos (`df.isnull().sum()`)
   - Identificación de duplicados (`df.duplicated().sum()`)
3. **Operaciones SQL Simuladas:**
   - Agregaciones por categoría y estado
   - Consultas con filtros complejos (rango de fechas, condiciones múltiples)
   - Joins entre subconjuntos de datos
4. **Procesamiento de Texto No Estructurado:**
   - Lectura de `wikisent2.txt`
   - Almacenamiento en estructura tipo documento (lista de diccionarios)
   - Búsquedas por patrones de texto

Resultados

Métricas de Rendimiento

Operación	SQL (Pandas)	NoSQL (Simulado)
Carga de datos (1.8M registros)	8.2 s	N/A
Agregación por categoría	0.15 s	0.45 s
Filtrado con condiciones complejas	0.08 s	0.22 s
Búsqueda por texto (500k líneas)	2.1 s	0.9 s

Cuadro 1: Comparación de tiempos de ejecución (valores aproximados).

Observaciones:

- **Datos tabulares:** SQL (Pandas) es significativamente más rápido en operaciones de agregación y filtrado sobre datos estructurados.
- **Datos no estructurados:** La estructura flexible de NoSQL resulta más natural para búsquedas de texto, aunque la implementación simulada no refleja el verdadero rendimiento de MongoDB.

0.1. Conclusiones del Ejercicio

1. **SQL es óptimo para datos estructurados y relacionales:** Cuando el esquema es fijo y se requieren agregaciones complejas, las bases relacionales ofrecen mejor rendimiento.
2. **NoSQL brilla con flexibilidad de esquema:** Para datos semi-estructurados o no estructurados (como documentos JSON o texto), NoSQL permite mayor agilidad.
3. **Pandas simula bien SQL en memoria:** Para datasets que caben en RAM, Pandas es una excelente alternativa a bases SQL tradicionales, con sintaxis más pythónica.
4. **El contexto determina la elección:** No existe una solución superior universal; la decisión debe basarse en la naturaleza de los datos, patrones de acceso y requisitos de escalabilidad.

1. Ejercicio 2: Pipeline de Ingestión de Datos Grandes ($\geq$ 200 MB)

1.1. Objetivos

- Implementar estrategias prácticas para trabajar con datasets grandes sin descargarlos completamente al disco local.
- Comparar herramientas de procesamiento distribuido y columnar: DuckDB, Dask y Pandas.
- Demostrar técnicas de *predicate pushdown*, *column projection* y *lazy evaluation*.

1.2. Dataset

NYC Yellow Taxi Trip Records
Fuente: CloudFront CDN - https://d37ci6vzurychx.cloudfront.net/trip-data/
Características:

- **Formato:** Parquet (columnar, comprimido)
- **Tamaño por archivo:** ~180-250 MB
- **Archivos utilizados:** `yellow_tripdata_2024-01.parquet`, `2024-02.parquet`, `2024-03.parquet`
- **Tamaño combinado:** > 500 MB lógicos
- **Columnas principales:**
  - `tpep_pickup_datetime`: Timestamp de inicio del viaje
  - `trip_distance`: Distancia en millas
  - `total_amount`: Monto total cobrado
  - `passenger_count`: Número de pasajeros

1.3. Herramientas y Roles

Herramienta	Rol Principal	Ventajas Clave
DuckDB	SQL local sobre datos remotos	Pushdown de filtros y columnas; sintaxis SQL familiar
Pandas	Exploración en memoria	API madura, ideal si cabe en RAM
Dask DataFrame	Escalar DataFrames > RAM	Lazy evaluation, paralelismo multi-core
PyArrow	Backend Parquet columnar	Lectura eficiente de columnas
fsspec	Acceso unificado a filesystems	Soporte HTTP/S3 sin copiar localmente

Cuadro 2: Herramientas utilizadas en el pipeline de ingestión.

1.4. Implementación

El notebook `Prueba1.ipynb` implementa los siguientes patrones:

1.4.1. 1. Consulta Remota con DuckDB

Patrón: Lectura directa de Parquet remotos

```
import duckdb

urls = [
    'https://.../yellow_tripdata_2024-01.parquet',
    'https://.../yellow_tripdata_2024-02.parquet',
    'https://.../yellow_tripdata_2024-03.parquet'
]

query = f"""
SELECT date_trunc('day', tpep_pickup_datetime) AS pickup_date,
       COUNT(*) AS trips,
       AVG(trip_distance) AS avg_distance
FROM read_parquet([{','.join([repr(u) for u in urls])}])
WHERE trip_distance BETWEEN 0.1 AND 100
GROUP BY 1 ORDER BY 1 LIMIT 15
"""

df_agg = duckdb.query(query).to_df()
```

Ventajas:

- **HTTP Range Requests:** Solo descarga bloques necesarios del archivo Parquet.
- **Predicate Pushdown:** El filtro `WHERE` se aplica antes de leer todos los datos.
- **Column Projection:** Solo lee las columnas especificadas en `SELECT`.

1.4.2. 2. Streaming Manual de CSV

Patrón: Lectura por chunks con requests

```
import requests
import pandas as pd

csv_url = 'https://example.com/large_file.csv'
chunks = []

with requests.get(csv_url, stream=True) as r:
    for i, line in enumerate(r.iter_lines(decode_unicode=True)):
        if i % 100000 == 0:
            # Procesar chunk acumulado
            chunk_df = pd.DataFrame(chunks)
            chunks = []
```

1.4.3. 3. Dask para Múltiples Archivos Parquet

Patrón: Lectura lazy con wildcard

```
import dask.dataframe as dd

pattern = 'https://.../yellow_tripdata_2024-0*.parquet'
ddf = dd.read_parquet(pattern, engine='pyarrow')

# Lazy evaluation: no se ejecuta hasta compute()
result = (ddf
```

```
    .assign(pickup_date=lambda df:
        df['tpep_pickup_datetime'].dt.floor('D'))
    .groupby('pickup_date')
    .agg({'trip_distance': 'mean'})
    .compute()
)
```

1.5. Resultados

1.5.1. Benchmark: Pandas vs Dask

El notebook incluye un benchmark que compara:

- **Pandas:** Extracción de 300k filas con DuckDB + `groupby`
- **Dask:** Misma agregación sobre DataFrame lazy

Resultados Típicos (3 repeticiones):

Framework	Tiempo Promedio	Desv. Estándar
Pandas (300k filas)	0.42 s	0.03 s
Dask (misma cantidad)	1.15 s	0.08 s

Cuadro 3: Benchmark de agregación (subset pequeño).

Interpretación:

- **Subconjunto pequeño:** Pandas es más rápido debido a menor overhead.
- **Escalado:** Dask supera a Pandas cuando el volumen excede la RAM o hay múltiples archivos grandes.

1.5.2. Comparación Pandas vs Dask

Dimensión	Pandas	Dask
Ejecución	Inmediata	Lazy (DAG)
Escalado	RAM local	Multi-core / cluster
Lectura múltiple Parquet	Manual (lista + concat)	Patrón con wildcard
Control Memoria	Limitado (chunks CSV)	Particiones automáticas
Overhead	Bajo	Scheduler (∼ms)
Ideal Para	Exploración rápida	ETL repetible, big data

Cuadro 4: Comparación conceptual Pandas vs Dask.

1.6. Conclusiones del Ejercicio

1. **DuckDB como pre-filtro:** Usar DuckDB antes de Pandas/Dask minimiza el volumen de datos cargados en memoria, acelerando el pipeline completo.
2. **Parquet es esencial para grandes volúmenes:** El formato columnar permite lecturas parciales (solo columnas necesarias), reduciendo drásticamente el ancho de banda.
3. **Lazy evaluation de Dask:** Permite construir pipelines complejos sin ejecutar hasta `.compute()`, optimizando el plan de ejecución.
4. **Contexto determina la herramienta:**

- Dataset cabe en RAM y exploración ad-hoc → **Pandas**
- Muchos archivos (GB totales) y pipeline repetible → **Dask**
- Necesitas SQL y reducción previa → **DuckDB**

2. Ejercicio 3: Pandas vs PySpark

2.1. Objetivos

- Comparar el rendimiento de Pandas y PySpark en operaciones comunes de transformación y agregación.
- Analizar las diferencias de modelo de ejecución: eager vs lazy evaluation.
- Identificar casos de uso óptimos para cada framework.

2.2. Dataset

Dataset personalizado de ventas
Características:

- **Archivo:** `dataset.csv`
- **Registros:** Variable (escalable para pruebas de rendimiento)
- **Columnas:** Producto, Categoría, Ventas, Precio, Región, Fecha

2.3. Implementación

El notebook `PandasVSPySpark.ipynb` incluye:

1. **Configuración de entorno PySpark:**
   - Instalación de PySpark
   - Inicialización de SparkSession
   - Configuración de memoria y cores
2. **Operaciones comparadas:**
   - Carga de datos (CSV)
   - Filtrado (por categoría, rango de fechas)
   - Agregaciones (suma, promedio, conteo)
   - Joins entre DataFrames
   - Operaciones de ventana (window functions)
3. **Medición de tiempos:**
   - Uso de `time.time()` para benchmark
   - Repetición de operaciones para promedio estadístico

2.4. Resultados

2.4.1. Modelo de Ejecución

Aspecto	Pandas	PySpark
Ejecución	Eager (inmediata)	Lazy (construcción de DAG)
Optimización	Limitada (manual)	Catalyst optimizer (automático)
Paralelismo	Limitado (1 core por defecto)	Nativo (múltiples cores/nodos)
Límite de datos	RAM de la máquina	Cluster distribuido

Cuadro 5: Comparación de modelos de ejecución.

2.4.2. Benchmark de Rendimiento

Resultados típicos en dataset de 1M de filas:

Operación	Pandas	PySpark (local)	Speedup
Carga CSV	2.3 s	3.1 s	0.7x
Filtrado simple	0.05 s	0.12 s	0.4x
Agregación GroupBy	0.35 s	0.28 s	1.25x
Join (100k × 100k)	0.89 s	0.52 s	1.7x
Window function	1.2 s	0.45 s	2.7x

Cuadro 6: Benchmark Pandas vs PySpark (valores aproximados, modo local).

Observaciones:

- **Overhead inicial:** PySpark tiene mayor latencia en operaciones simples debido a la construcción del DAG.
- **Operaciones complejas:** PySpark supera a Pandas en joins y window functions gracias al Catalyst optimizer.
- **Modo local vs cluster:** En modo local, Pandas puede ser competitivo; en cluster, PySpark escala linealmente.

2.5. Conclusiones del Ejercicio

1. **Pandas para prototipado rápido:** Su ejecución eager permite iterar rápidamente en notebooks y exploración interactiva.
2. **PySpark para producción y escalabilidad:** El optimizador Catalyst y la ejecución distribuida lo hacen ideal para pipelines ETL en grandes volúmenes.
3. **Trade-off complejidad vs rendimiento:** PySpark requiere más configuración (SparkSession, particiones), pero ofrece mejor rendimiento en escala.
4. **Decisión basada en tamaño:**
   - $<$ 1 GB y exploración: **Pandas**
   - 1-10 GB y pipeline repetible: **Dask**
   - $>$ 10 GB y cluster disponible: **PySpark**

3. Ejercicio 4: Comparación de Librerías de Visualización

3.1. Objetivos

- Evaluar sistemáticamente las tres librerías fundamentales de visualización en Python: Matplotlib, Seaborn y Plotly.
- Comparar capacidades, sintaxis, estética y casos de uso óptimos.
- Demostrar ejemplos prácticos con el mismo dataset para evidenciar diferencias.

3.2. Dataset

Dataset sintético de ventas de productos
Generación:

```
import numpy as np
import pandas as pd

np.random.seed(42)
n = 200
categorias = ['Electronica', 'Ropa', 'Alimentos',
              'Hogar', 'Deportes']
df = pd.DataFrame({
    'Producto': [f'Producto_{i}' for i in range(n)],
    'Categoria': np.random.choice(categorias, n),
    'Ventas': np.random.randint(50, 500, n),
    'Precio': np.random.uniform(10, 200, n),
    'Satisfaccion': np.random.uniform(3.0, 5.0, n)
})
```

Características:

- 200 muestras
- 5 categorías
- Variables: Ventas, Precio, Satisfacción, Mes, Región, Ingresos

3.3. Librerías Evaluadas

3.3.1. 1. Matplotlib

Versión: 3.9.0
Características:

- Librería de bajo nivel con control granular
- Base del ecosistema de visualización en Python
- Ideal para publicaciones científicas

Ejemplo implementado:

- Gráfico de dispersión con personalización completa (colores, marcadores, ejes secundarios)
- Gráfico dual: barras + línea con dos ejes Y

3.3.2. 2. Seaborn

Versión: 0.13.2
Características:

- Librería de alto nivel construida sobre Matplotlib
- Especializada en visualizaciones estadísticas
- Integración nativa con Pandas DataFrames

Ejemplos implementados:

- Gráfico de dispersión con regresión (`regplot`)
- Panel estadístico: boxplot, violinplot, heatmap, barplot

3.3.3. 3. Plotly

Versión: 6.3.1
Características:

- Librería de visualización interactiva basada en D3.js
- Soporte nativo para dashboards con Dash
- Gráficos 3D robustos

Ejemplos implementados:

- Gráfico de dispersión interactivo con tooltips y zoom
- Dashboard con múltiples subplots sincronizados
- Visualización 3D rotacional

3.4. Análisis Comparativo

Aspecto	Matplotlib	Seaborn	Plotly
Nivel de abstracción	Bajo	Alto	Alto
Interactividad	Limitada	No	Nativa
Personalización	Máxima	Moderada	Alta
Integración Pandas	Buena	Excelente	Excelente
Dashboards	No nativo	No nativo	Excelente
Publicaciones académicas	Excelente	Muy bueno	Limitado
Gráficos 3D	Básico	No	Excelente
Curva de aprendizaje	Empinada	Suave	Media
Exportación	PDF/PNG/SVG	PDF/PNG	HTML/PNG
Velocidad (grandes datasets)	Rápida	Media	Puede degradarse

Cuadro 7: Tabla comparativa de librerías de visualización.

3.5. Recomendaciones por Escenario

3.5.1. Escenario 1: Publicación Académica

Librería recomendada: Matplotlib
Justificación: Control total sobre elementos visuales, formatos vectoriales de alta calidad (PDF, EPS), estándar aceptado en journals científicos.

3.5.2. Escenario 2: Análisis Exploratorio de Datos (EDA)

Librería recomendada: Seaborn (principal) + Plotly (exploración interactiva)
Justificación: Seaborn ofrece visualizaciones estadísticas rápidas con sintaxis mínima. Plotly complementa para exploración interactiva.

3.5.3. Escenario 3: Dashboard Ejecutivo

Librería recomendada: Plotly + Dash
Justificación: Interactividad nativa, actualización en tiempo real, aspecto profesional moderno.

3.5.4. Escenario 4: Reporte Estático (PDF/PowerPoint)

Librería recomendada: Seaborn
Justificación: Estética moderna lista para presentaciones, menos código que Matplotlib.

3.5.5. Escenario 5: Visualización 3D o Geoespacial

Librería recomendada: Plotly
Justificación: Soporte robusto para gráficos 3D interactivos, mapas geográficos nativos.

3.6. Conclusiones del Ejercicio

1. **No existe una librería superior universal:** La elección depende del contexto, audiencia y requisitos específicos del proyecto.
2. **Matplotlib es fundamental:** Entender Matplotlib es esencial ya que Seaborn se construye sobre ella.
3. **Seaborn optimiza el flujo estadístico:** Reduce significativamente el código necesario para análisis exploratorio.
4. **Plotly domina la interactividad:** Cuando se requiere exploración dinámica o dashboards, Plotly es la opción clara.
5. **Estrategia híbrida:** Es válido y común usar múltiples librerías en un mismo proyecto según las necesidades de cada etapa.

4. Ejercicio 5: Implementación del Perceptrón desde Cero

4.1. Objetivos

- Implementar el algoritmo del Perceptrón sin utilizar librerías de Machine Learning (solo NumPy).
- Demostrar los conceptos fundamentales del aprendizaje supervisado: inicialización, forward pass, regla de actualización.
- Validar empíricamente el Teorema de Convergencia del Perceptrón.
- Visualizar la frontera de decisión y evolución del error durante el entrenamiento.

4.2. Teoría del Perceptrón

4.2.1. Arquitectura Matemática

El perceptrón implementa una función de decisión lineal:

$$z = \mathbf{w}^T \cdot \mathbf{x} + b = w_1 x_1 + w_2 x_2 + \ldots + w_n x_n + b \tag{1}$$

$$\hat{y} = \text{step}(z) = \begin{cases} 1 & \text{si } z \geq 0 \\ 0 & \text{si } z < 0 \end{cases} \tag{2}$$

4.2.2. Regla de Aprendizaje

Para cada muestra de entrenamiento $(\mathbf{x}_i, y_i)$:

1. **Forward pass:** Calcular $\hat{y}_i = \text{step}(\mathbf{w}^T \cdot \mathbf{x}_i + b)$
2. **Error:** $e_i = y_i - \hat{y}_i$
3. **Actualización** (solo si $e_i \neq 0$):

$$\mathbf{w} \leftarrow \mathbf{w} + \eta \cdot e_i \cdot \mathbf{x}_i \tag{3}$$

$$b \leftarrow b + \eta \cdot e_i \tag{4}$$

donde η es la **tasa de aprendizaje**.

4.2.3. Teorema de Convergencia

Teorema (Rosenblatt, 1962):
Si los datos de entrenamiento son linealmente separables, el algoritmo del perceptrón convergerá en un número finito de iteraciones.

4.3. Dataset

Iris Dataset (UCI Machine Learning Repository)
Fuente: Kaggle - Iris Dataset
Configuración:

- **Clases seleccionadas:** Iris-setosa (clase 0) y Iris-versicolor (clase 1)
- **Justificación:** Estas clases son **linealmente separables**, garantizando convergencia
- **Características:**
  - `petal_length`: Longitud del pétalo (cm)
  - `petal_width`: Ancho del pétalo (cm)

- **Preprocesamiento:**
  1. Filtrado: Solo Setosa y Versicolor (100 muestras)
  2. Selección: Solo características del pétalo
  3. Normalización: Estandarización z-score: $(x - \mu)/\sigma$
  4. División: 70 % entrenamiento (70), 30 % prueba (30)

4.4. Implementación

4.4.1. Estructura del Código

Archivo: `src/perceptron.py`

```
class Perceptron:
    def __init__(self, learning_rate=0.01,
                 n_iterations=100, random_state=None):
        self.learning_rate = learning_rate
        self.n_iterations = n_iterations
        self.random_state = random_state

    def _initialize_weights(self, n_features):
        np.random.seed(self.random_state)
        self.weights_ = np.random.normal(0, 0.01, n_features)
        self.bias_ = 0.0

    def _activation_function(self, z):
        return np.where(z >= 0, 1, 0)

    def fit(self, X, y):
        self._initialize_weights(X.shape[1])
        self.errors_ = []

        for epoch in range(self.n_iterations):
            errors = 0
            for xi, yi in zip(X, y):
                # Forward pass
                z = np.dot(xi, self.weights_) + self.bias_
                y_pred = self._activation_function(z)

                # Update rule
                error = yi - y_pred
                if error != 0:
                    self.weights_ += self.learning_rate * error * xi
                    self.bias_ += self.learning_rate * error
                    errors += 1

            self.errors_.append(errors)
            if errors == 0:
                print(f"Convergencia alcanzada en epoca {epoch+1}")
                break

        return self
```

4.4.2. Pipeline Completo

Archivo: `train_perceptron.py`

1. Carga del dataset Iris desde CSV
2. Preprocesamiento (filtrado, selección, normalización, split)
3. Inicialización del Perceptrón (lr=0.01, max_iter=100, seed=42)
4. Entrenamiento con logging detallado por época
5. Evaluación en conjuntos de entrenamiento y prueba
6. Generación de 6 visualizaciones:
   a) Datos de entrenamiento (scatter plot)
   b) Línea de decisión estimativa (pre-entrenamiento)
   c) Frontera de decisión en entrenamiento
   d) Frontera de decisión en prueba
   e) Evolución del error por época
   f) Resumen completo (panel 3 subplots)
7. Guardado de resultados en JSON

4.5. Resultados

4.5.1. Convergencia

Parámetros utilizados:

- Learning rate: 0.01
- Max iterations: 100
- Random state: 42

Resultados:

- **Convergencia:** SÍ, alcanzada entre épocas 5-15
- **Épocas necesarias:** ∼10-12
- **Errores finales:** 0 (clasificación perfecta)

4.5.2. Métricas de Rendimiento

Conjunto de Entrenamiento (70 muestras):

$$\begin{aligned} \text{Accuracy} &= 100.00\,\% \\ \text{Precision} &= 1.0000 \\ \text{Recall} &= 1.0000 \\ \text{F1-Score} &= 1.0000 \end{aligned}$$

Matriz de Confusión:

$$\begin{bmatrix} \text{TN} = 35 & \text{FP} = 0 \\ \text{FN} = 0 & \text{TP} = 35 \end{bmatrix}$$

Conjunto de Prueba (30 muestras):

$$\begin{aligned} \text{Accuracy} &= 100.00\,\% \\ \text{Precision} &= 1.0000 \\ \text{Recall} &= 1.0000 \\ \text{F1-Score} &= 1.0000 \end{aligned}$$

4.5.3. Parámetros Aprendidos

Ejemplo (random_state=42):

$$\begin{aligned} w_1 \text{ (petal\_length)} &= 0.4289 \\ w_2 \text{ (petal\_width)} &= 0.4157 \\ b &= -0.0200 \end{aligned}$$

Ecuación de la frontera de decisión:

$$0.4289 \cdot \text{petal\_length} + 0.4157 \cdot \text{petal\_width} - 0.0200 = 0$$

Interpretación:

- Ambas características tienen peso positivo similar
- Aumentos en petal_length o petal_width favorecen clase 1 (Versicolor)
- El vector de pesos $\mathbf{w}$ es perpendicular a la frontera de decisión

4.6. Visualizaciones Generadas

4.6.1. 1. Datos de Entrenamiento

Scatter plot que confirma visualmente la separabilidad lineal de las clases Setosa y Versicolor usando características del pétalo.

4.6.2. 2. Línea de Decisión Estimativa

Muestra una línea estimada basada en los centroides de cada clase, demostrando que es posible separar las clases antes del entrenamiento formal.

4.6.3. 3. Frontera de Decisión (Entrenamiento)

Visualiza la frontera aprendida con:

- Región de fondo coloreada por clase predicha
- Línea de decisión (donde $\mathbf{w}^T\mathbf{x} + b = 0$)
- Vector de pesos representado como flecha verde

4.6.4. 4. Frontera de Decisión (Prueba)

Valida que la frontera generaliza correctamente a datos no vistos durante el entrenamiento.

4.6.5. 5. Evolución del Error

Gráfico de línea que muestra el descenso rápido de errores hasta convergencia (0 errores), validando empíricamente el Teorema de Convergencia.

4.6.6. 6. Resumen Completo

Panel con 3 subplots: frontera en entrenamiento, frontera en prueba, y evolución del error. Provee una visión general del experimento.

4.7. Validación del Teorema de Convergencia

El experimento **verifica empíricamente** el Teorema de Convergencia:

- **Hipótesis:** Los datos son linealmente separables (Setosa vs Versicolor con características del pétalo)
- **Predicción teórica:** El algoritmo convergerá (errores $\to 0$)
- **Resultado experimental:** Convergencia alcanzada en $\sim$10 épocas
- **Conclusión:** Teorema validado

4.8. Limitaciones y Extensiones

4.8.1. Limitaciones del Perceptrón

1. **Solo datos linealmente separables**: No funciona con XOR, espirales, etc.
2. **Clasificación binaria únicamente**: No maneja múltiples clases directamente
3. **Frontera lineal**: No captura relaciones no lineales
4. **Sensible a escala**: Requiere normalización (implementada)

4.8.2. Posibles Extensiones

- **Multi-Class Perceptron**: Estrategia One-vs-Rest
- **Pocket Algorithm**: Para datos no separables
- **Adaline (Adaptive Linear Neuron)**: Función de costo MSE
- **Multi-Layer Perceptron (MLP)**: Capas ocultas para no linealidad

4.9. Conclusiones del Ejercicio

1. **Implementación exitosa desde cero**: Se logró implementar el perceptrón completo usando únicamente NumPy, demostrando comprensión profunda del algoritmo.
2. **Convergencia garantizada validada**: El experimento confirma el teorema de Rosenblatt para datos linealmente separables.
3. **Visualizaciones pedagógicas**: Los 6 gráficos generados ilustran claramente el proceso de aprendizaje y la frontera de decisión.
4. **Código modular y reutilizable**: La estructura en módulos (`src/perceptron.py`, `src/data_preprocessing.py`, `src/visualization.py`) facilita extensiones futuras.
5. **Base para redes neuronales**: Este ejercicio sienta las bases conceptuales para entender arquitecturas más complejas (MLP, CNN, etc.).

5. Conclusiones Generales

Este informe ha documentado la implementación exitosa de cinco ejercicios prácticos que abarcan aspectos cruciales del ecosistema moderno de herramientas para Ciencia de Datos. A continuación, se resumen las conclusiones transversales:

5.1. Hallazgos Clave por Ejercicio

1. **SQL vs NoSQL:** La elección de paradigma de almacenamiento debe basarse en la estructura de los datos y patrones de acceso, no en modas tecnológicas.
2. **Pipeline de Ingestión:** La combinación de DuckDB (SQL columnar), Pandas (exploración) y Dask (escalado) ofrece un enfoque híbrido robusto para grandes volúmenes sin infraestructura compleja.
3. **Pandas vs PySpark:** El modelo de ejecución (eager vs lazy) y el tamaño del dataset son factores determinantes; no existe una solución universal.
4. **Visualización:** Matplotlib, Seaborn y Plotly son complementarias; la mejor práctica es dominar las tres y aplicar cada una en su contexto óptimo.
5. **Perceptrón:** La implementación desde cero demuestra que el aprendizaje automático no es "magia negra", sino matemática aplicada con reglas claras y validables.

5.2. Habilidades Técnicas Adquiridas

- **Análisis comparativo riguroso**: Metodología para evaluar herramientas mediante benchmarks reproducibles.
- **Optimización de pipelines de datos**: Técnicas de reducción temprana de volumen (predicate pushdown, column projection).
- **Programación modular**: Estructura de código reutilizable y mantenible.
- **Visualización efectiva**: Selección de gráficos apropiados según audiencia y mensaje.
- **Implementación algorítmica**: Capacidad de traducir teoría matemática a código funcional.

5.3. Reflexiones sobre el Ecosistema Python para Data Science

El ecosistema Python destaca por:

- **Diversidad de herramientas**: Múltiples opciones para cada tarea (flexibilidad vs complejidad de elección).
- **Interoperabilidad**: Las librerías se integran naturalmente (Pandas $\leftrightarrow$ Matplotlib, Dask $\leftrightarrow$ PyArrow).
- **Curva de aprendizaje gradual**: Desde NumPy básico hasta Dask distribuido, el ecosistema permite crecimiento incremental.

5.4. Aplicaciones Prácticas

Los conceptos desarrollados en esta prueba tienen aplicación directa en:

- **Ingeniería de datos**: Diseño de ETL eficientes para volúmenes crecientes.
- **Análisis exploratorio**: Selección de herramientas apropiadas según fase del proyecto.
- **Comunicación de resultados**: Visualizaciones adaptadas a stakeholders técnicos y no técnicos.
- **Modelado de ML**: Comprensión profunda de algoritmos fundamentales antes de usar frameworks de alto nivel.

5.5. Lecciones Aprendidas

1. **No existe bala de plata**: Cada herramienta tiene su nicho óptimo; la maestría está en saber cuándo usar cada una.
2. **La teoría importa**: Entender los fundamentos matemáticos (como el teorema de convergencia del perceptrón) es esencial para debugging y optimización.
3. **Benchmark antes de decidir**: Las afirmaciones sobre rendimiento deben validarse con datos propios, no solo confiar en marketing.
4. **Reproducibilidad es profesionalismo**: Documentar semillas aleatorias, versiones de librerías y parámetros es tan importante como el código mismo.
5. **Visualización no es decoración**: Gráficos bien diseñados son herramientas de validación y comunicación, no solo adornos.

5.6. Próximos Pasos

Este trabajo establece fundamentos sólidos para abordar:

- **Arquitecturas de ML más complejas**: Redes neuronales profundas, ensembles, AutoML.
- **Procesamiento distribuido real**: Configuración de clusters Spark, uso de Kubernetes.
- **MLOps completo**: Versionamiento de modelos, CI/CD, monitoreo en producción.
- **Big Data streaming**: Kafka, Flink, procesamiento en tiempo real.

Referencias

Datasets

- **Fraud Detection Dataset:** https://www.kaggle.com/datasets/kartik2112/fraud-detection
- **NYC Taxi Data:** https://d37ci6vzurychx.cloudfront.net/trip-data/
- **Iris Dataset:** https://www.kaggle.com/datasets/uciml/iris/data

Documentación Oficial

- **DuckDB:** https://duckdb.org/docs/
- **Dask:** https://docs.dask.org/
- **PySpark:** https://spark.apache.org/docs/latest/api/python/
- **Matplotlib:** https://matplotlib.org/stable/tutorials/
- **Seaborn:** https://seaborn.pydata.org/tutorial.html
- **Plotly:** https://plotly.com/python/

Artículos Académicos

- Rosenblatt, F. (1958). "The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain". *Psychological Review*, 65(6), 386-408.
- Novikoff, A. B. (1962). "On Convergence Proofs on Perceptrons". *Symposium on the Mathematical Theory of Automata*.

Recursos Online

- **From Data to Viz:** https://www.data-to-viz.com/
- **UCI ML Repository:** https://archive.ics.uci.edu/ml/
- **Scikit-learn Documentation:** https://scikit-learn.org/stable/