UNIVERSIDAD DE SANTIAGO DE CHILE
Facultad de Ingeniería
Departamento de Ingeniería Informática

Manual del Profesor

Clase 04: Pipeline de Desarrollo Geoespacial

Geoinformática - Segundo Semestre 2025

Prof. Francisco Parra O.

2 de septiembre de 2025

Índice

1. Resumen Ejecutivo

1.1. Objetivo de la Clase

Esta clase tiene como objetivo proporcionar a los estudiantes una comprensión integral de los pipelines de desarrollo geoespacial, combinando principios arquitectónicos sólidos con implementaciones prácticas. Se enfoca en construir soluciones escalables y mantenibles desde la conceptualización hasta el despliegue.

1.2. Resultados de Aprendizaje Esperados

Al finalizar la clase, los estudiantes serán capaces de:

- Aplicar principios SOLID al diseño de pipelines geoespaciales
- Comprender y aplicar teoría de grafos en redes viales
- Seleccionar paradigmas de procesamiento apropiados (batch, stream, micro-batch)
- Diseñar arquitecturas de microservicios geoespaciales
- Implementar estrategias de testing y monitoreo
- Identificar y resolver desafíos comunes en pipelines geoespaciales
- Estructurar profesionalmente un proyecto geoespacial
- Conectarse y obtener datos de APIs reales (OSM, Google Maps, IDE Chile)
- Implementar análisis espaciales complejos (clustering, interpolación)
- Optimizar consultas y operaciones para grandes volúmenes de datos
- Desplegar una aplicación geoespacial escalable

1.3. Duración y Estructura

- **Duración total**: 90 minutos
- **Formato**: Presencial con demos en vivo
- **Metodología**: Learning by doing - cada concepto se acompaña de código ejecutable

2. Preparación Pre-Clase

2.1. Requerimientos Técnicos

Importante

Es CRUCIAL verificar estos elementos antes de la clase para evitar retrasos técnicos.

- **Software instalado**:
  - Python 3.9+ con ambiente virtual preparado
  - PostgreSQL con PostGIS habilitado
  - Docker Desktop funcionando
  - VS Code o Jupyter Lab

- **Cuentas y API Keys**:
  - Cuenta Google Cloud con $300 de crédito gratuito
  - API Key de OpenWeatherMap (gratuita)
  - Cuenta en GitHub
- **Datasets de prueba**:
  - Shapefile de comunas de Chile (IDE Chile)
  - CSV con direcciones de Santiago para geocodificar
  - GeoJSON de red de metro de Santiago

2.2. Ambiente de Desarrollo

Crear un ambiente Python con todas las dependencias:

```
# Crear ambiente virtual
python -m venv geo_env
source geo_env/bin/activate  # Linux/Mac
# o
geo_env\Scripts\activate  # Windows

# Instalar dependencias
pip install geopandas pandas numpy
pip install folium streamlit fastapi uvicorn
pip install osmnx networkx scikit-learn
pip install sqlalchemy psycopg2-binary
pip install python-dotenv requests
pip install dask[complete] dask-geopandas
pip install pykrige rtree
```

3. Marco Teórico y Conceptual

3.1. Principios SOLID en Pipelines Geoespaciales

3.1.1. Single Responsibility Principle (SRP)

Cada componente del pipeline debe tener una única responsabilidad bien definida:

- Módulo de geocodificación: solo convierte direcciones a coordenadas
- Servicio de routing: solo calcula rutas óptimas
- Componente de visualización: solo renderiza mapas

Ejemplo práctico: En lugar de tener una clase `GeoProcessor` que hace todo, separar en `GeoValidator`, `GeoTransformer`, y `GeoAnalyzer`.

3.1.2. Open/Closed Principle (OCP)

El sistema debe ser abierto para extensión pero cerrado para modificación:

- Usar interfaces para definir contratos
- Permitir agregar nuevas fuentes de datos sin modificar código existente
- Implementar plugins para nuevos tipos de análisis

3.1.3. Liskov Substitution Principle (LSP)

Las implementaciones deben ser intercambiables:

- Diferentes geocoders (Google, Nominatim, Mapbox) con misma interfaz
- Múltiples backends de almacenamiento (PostGIS, MongoDB, BigQuery)
- Proveedores de tiles intercambiables

3.1.4. Interface Segregation Principle (ISP)

No forzar dependencias innecesarias:

- APIs específicas para diferentes tipos de clientes
- Separar interfaces de lectura y escritura
- Métodos granulares en lugar de interfaces monolíticas

3.1.5. Dependency Inversion Principle (DIP)

Depender de abstracciones, no de implementaciones concretas:

- Inyección de dependencias para servicios
- Configuración externa (archivos YAML, variables de entorno)
- Factories para crear objetos complejos

3.2. Teoría de Grafos Aplicada a Redes Viales

3.2.1. Conceptos Fundamentales

- **Nodos (Vértices):** Representan intersecciones, puntos de interés
- **Aristas (Edges):** Representan segmentos de calle, conexiones
- **Pesos:** Distancia, tiempo de viaje, costo, tráfico
- **Dirección:** Calles de uno o doble sentido

3.2.2. Métricas de Red

- **Centralidad de grado:** Número de conexiones de un nodo
- **Centralidad de cercanía:** Distancia promedio a otros nodos
- **Centralidad de intermediación:** Frecuencia en caminos más cortos
- **Coeficiente de clustering:** Conectividad local

3.2.3. Algoritmos Clave

- **Dijkstra:** Camino más corto desde un origen
- **A*:** Dijkstra con heurística para mayor eficiencia
- **Bellman-Ford:** Maneja pesos negativos
- **Floyd-Warshall:** Todos los caminos más cortos
- **Kruskal/Prim:** Árbol de expansión mínima

3.3. Paradigmas de Procesamiento Geoespacial

3.3.1. Batch Processing

Características:

- Procesamiento de grandes volúmenes en bloques
- Alta latencia aceptable
- Optimizado para throughput

Casos de uso:

- Análisis histórico de patrones de movilidad
- Generación de reportes mensuales
- Procesamiento de imágenes satelitales

Tecnologías: Apache Spark, PostGIS batch operations, GDAL

3.3.2. Stream Processing

Características:

- Procesamiento continuo de datos en tiempo real
- Baja latencia crítica
- Optimizado para respuesta rápida

Casos de uso:

- Tracking GPS en tiempo real
- Alertas de geofencing
- Detección de anomalías espaciales

Tecnologías: Apache Kafka, Apache Flink, Redis Streams

3.3.3. Micro-batch Processing

Características:

- Híbrido entre batch y stream
- Procesamiento en ventanas de tiempo pequeñas
- Balance entre latencia y throughput

Casos de uso:

- Actualización de dashboards cada minuto
- Agregaciones temporales de sensores
- Cálculo de métricas de tráfico

Tecnologías: Spark Streaming, Storm Trident

3.4. Testing y Calidad en Pipelines Geoespaciales

3.4.1. Pirámide de Testing

- **Unit Tests (70 %):** Pruebas de funciones individuales
  - Validación de geometrías
  - Transformaciones de coordenadas
  - Cálculos de distancia
- **Integration Tests (20 %):** Pruebas de componentes conectados
  - Conexión a PostGIS
  - Llamadas a APIs externas
  - Pipeline ETL completo
- **System Tests (8 %):** Pruebas del sistema completo
  - Flujo completo de usuario
  - Rendimiento bajo carga
  - Recuperación ante fallos
- **E2E Tests (2 %):** Pruebas de extremo a extremo
  - Escenarios reales de usuario
  - Pruebas en ambiente de staging

3.4.2. Tests Específicos Geoespaciales

- **Validación de geometrías:** Polígonos cerrados, sin auto-intersecciones
- **Proyecciones correctas:** Verificar CRS antes y después
- **Topología consistente:** Nodos conectados, sin gaps
- **Precisión espacial:** Tolerancias y redondeo apropiados
- **Cobertura de área:** Verificar límites geográficos

3.5. Monitoreo y Observabilidad

3.5.1. Los Tres Pilares de la Observabilidad

- **Logs:** Eventos discretos con timestamp
  - Errores de geocodificación
  - Tiempos de respuesta de APIs
  - Validaciones fallidas
- **Métricas:** Valores numéricos agregados
  - Features procesadas por segundo
  - Latencia P50, P95, P99
  - Tasa de error por servicio
- **Traces:** Flujo de requests a través del sistema

- Path completo de una geocodificación
- Cuellos de botella en el pipeline
- Dependencias entre servicios

3.5.2. Métricas Clave para Pipelines Geoespaciales

- **Latencia de procesamiento:** Tiempo por geometría
- **Throughput:** Features/segundo, MB/segundo
- **Tasa de error:** Geometrías inválidas, timeouts
- **Uso de recursos:** CPU, RAM, almacenamiento, ancho de banda
- **Cache hit rate:** Eficiencia del cacheo espacial
- **Precisión de geocoding:** Confianza en resultados

3.6. Desafíos Comunes y Soluciones

3.6.1. Desafío 1: Volumen de Datos

Problema:

- Terabytes de imágenes satelitales
- Millones de puntos GPS por día
- Datasets que no caben en memoria

Soluciones:

- Particionamiento espacial (QuadTree, GeoHash)
- Procesamiento paralelo con Dask/Spark
- Streaming en lugar de batch
- Compresión y formatos eficientes (Parquet, COG)

3.6.2. Desafío 2: Heterogeneidad de Datos

Problema:

- Múltiples formatos (Shapefile, GeoJSON, KML, etc.)
- Diferentes sistemas de coordenadas
- Calidad variable de datos

Soluciones:

- ETL robusto con validación
- Estandarización a formato común
- Transformación automática de CRS
- Data quality scoring

3.6.3. Desafío 3: Tiempo Real

Problema:

- Actualizaciones constantes de sensores
- Baja latencia requerida (¡100ms)
- Consistencia eventual vs inmediata

Soluciones:

- Stream processing (Kafka, Flink)
- Caché distribuido (Redis, Hazelcast)
- Arquitectura CQRS
- WebSockets para push notifications

3.6.4. Desafío 4: Calidad de Datos

Problema:

- Geometrías inválidas o corruptas
- Datos faltantes o incompletos
- Duplicados y outliers

Soluciones:

- Validación automática con tolerancias
- Reparación de geometrías (buffer(0))
- Imputación espacial (Kriging, IDW)
- Detección de anomalías con ML

3.7. Patrones Arquitectónicos para Escalabilidad

3.7.1. Lambda Architecture

Componentes:

- **Batch Layer:** Procesamiento completo y preciso
- **Speed Layer:** Procesamiento rápido pero aproximado
- **Serving Layer:** Combina resultados de ambas capas

Aplicación Geoespacial:

- Batch: Análisis histórico de movilidad
- Speed: Tracking en tiempo real
- Serving: Dashboard unificado

3.7.2. Kappa Architecture

Concepto:

- Todo es un stream
- Reprocesar desde el log si necesario
- Simplifica la arquitectura (una sola capa)

Aplicación Geoespacial:

- Stream de eventos de sensores IoT
- Actualizaciones de posición GPS
- Cambios en OpenStreetMap

3.7.3. Event Sourcing

Principio:

- Guardar eventos, no estados
- Estado actual = suma de todos los eventos
- Permite "time travelz auditoría completa

Aplicación Geoespacial:

- Historial de movimientos de vehículos
- Cambios en el uso del suelo
- Evolución de la red vial

3.8. Estrategias de Escalamiento

3.8.1. Escalamiento Vertical

Cuándo aplicar:

- Base de datos PostGIS principal
- Cálculos complejos que requieren todos los datos
- Operaciones que no se pueden paralelizar

Límites:

- Costo exponencial
- Límite físico del hardware
- Single point of failure

3.8.2. Escalamiento Horizontal

Cuándo aplicar:

- APIs stateless
- Procesamiento paralelo de tiles
- Cache distribuido
- Microservicios independientes

Consideraciones:

- Particionamiento de datos (sharding)
- Consistencia eventual
- Overhead de coordinación

4. Guión Detallado de la Clase

4.1. Introducción (5 minutos)

Demo en vivo

Mostrar un dashboard geoespacial en producción como ejemplo de lo que construirán.

[00:00 - 00:02] Apertura:

"Buenos días. En la clase anterior vimos los fundamentos de datos geoespaciales. Hoy vamos directo a la práctica: cómo construir una solución geoespacial real de principio a fin."

[00:02 - 00:05] Motivación:

"Varios de ustedes están trabajando en proyectos como valoración inmobiliaria o optimización de rutas. Hoy aprenderán exactamente cómo implementar estas soluciones profesionalmente. Vean este dashboard [MOSTRAR DEMO] - al final de la clase sabrán cómo construir algo así."

4.2. Sección 1: Arquitectura y Principios (20 minutos)

[00:05 - 00:10] Aplicación de SOLID:

Demo en vivo

Mostrar diagrama UML de arquitectura siguiendo principios SOLID.

."Antes de escribir código, entendamos la arquitectura. Un pipeline mal diseñado es deuda técnica desde el día uno."

Ejemplos concretos para cada principio:

- **SRP:** ."Este servicio SOLO geocodifica, nada más."
- **OCP:** "Podemos agregar Mapbox sin tocar el código existente."
- **LSP:** Çualquier geocoder funciona igual: dirección entra, coordenadas salen."

- **ISP:** .El frontend no necesita saber de PostGIS.”
- **DIP:** Çambiamos de PostGIS a MongoDB solo cambiando configuración.”

[00:10 - 00:15] Teoría de Grafos para Redes:

```
# Conceptos de grafos aplicados
import networkx as nx
import osmnx as ox

# El grafo representa la red vial
G = ox.graph_from_place("Las Condes, Santiago", network_type='drive')

# Nodos = intersecciones
print(f"Intersecciones: {len(G.nodes)}")

# Aristas = calles
print(f"Segmentos de calle: {len(G.edges)}")

# M tricas de red
centrality = nx.betweenness_centrality(G)
top_nodes = sorted(centrality.items(), key=lambda x: x[1], reverse=True)[:5]
print("Intersecciones m s importantes (mayor flujo de tr fico):")
for node, score in top_nodes:
    print(f"  Nodo {node}: {score:.3f}")
```

[00:15 - 00:20] Paradigmas de Procesamiento:
Comparación práctica:

- **Batch:** ”Procesamos todos los datos de ayer a las 2 AM”
- **Stream:** ”Procesamos cada dato cuando llega, sin esperar”
- **Micro-batch:** ”Procesamos cada minuto lo que llegó”

4.3. Sección 2: Flujo de Trabajo Geoespacial (15 minutos)

[00:05 - 00:10] Pipeline completo:

> Demo en vivo
>
> Crear estructura de carpetas en tiempo real usando terminal.

```
# Demostraci n en vivo
mkdir proyecto_geo
cd proyecto_geo
mkdir -p data/{raw,processed,cache}
mkdir -p src/{etl,analysis,api,visualization}
mkdir -p tests config docker notebooks docs
touch config/config.yaml
touch src/__init__.py
touch README.md .gitignore .env
```

Explicar cada carpeta mientras se crea:

- `data/raw`: .Aquí van los datos originales, NUNCA los modificamos”
- `data/processed`: ”Datos limpios y listos para análisis”
- `src/etl`: ”Scripts de extracción y transformación”
- `config`: ”Toda la configuración centralizada”

[00:10 - 00:15] Control de versiones con DVC:

> **Tip**
>
> Si los estudiantes no conocen DVC, hacer analogía con Git LFS pero más poderoso.

```
# Demo DVC
pip install dvc
dvc init
dvc add data/comunas_santiago.gpkg
git add data/comunas_santiago.gpkg.dvc .gitignore
git commit -m "Add comunas dataset"

# Explicar el archivo .dvc generado
cat data/comunas_santiago.gpkg.dvc
```

Preguntas frecuentes en esta sección:

- "¿Por qué no usar Git directamente?"→ Git no maneja bien archivos ¿100MB
- "¿Qué pasa con las credenciales?"→ Usar .env y NUNCA commitear

4.4. Sección 2: Conexión a Fuentes de Datos Reales (20 minutos)

[00:15 - 00:20] OpenStreetMap con OSMnx:

> **Importante**
>
> Esta demo requiere conexión a internet estable. Tener datos cached como backup.

```
# notebook: 01_osm_data.ipynb
import osmnx as ox
import geopandas as gpd
import matplotlib.pyplot as plt

# Configurar cache para evitar re-descargas
ox.config(use_cache=True, log_console=True)

# Demo 1: Obtener red vial
print("Descargando red vial de Las Condes...")
G = ox.graph_from_place("Las Condes, Santiago, Chile",
                        network_type='drive')
print(f"Nodos: {len(G.nodes)}, Aristas: {len(G.edges)}")

# Convertir a GeoDataFrame
nodes, edges = ox.graph_to_gdfs(G)
edges[['name', 'highway', 'maxspeed', 'length']].head()

# Demo 2: Obtener POIs
print("Buscando hospitales...")
tags = {'amenity': 'hospital'}
hospitales = ox.geometries_from_place("Santiago, Chile", tags)
print(f"Encontrados: {len(hospitales)} hospitales")

# Visualizar
fig, ax = plt.subplots(figsize=(12, 8))
edges.plot(ax=ax, color='gray', linewidth=0.5)
hospitales.plot(ax=ax, color='red', markersize=100)
plt.title("Red vial y hospitales en Santiago")
plt.show()
```

[00:20 - 00:25] PostGIS - Base de datos espacial:

Demo en vivo

Mostrar pgAdmin o DBeaver con la base de datos ya configurada.

```python
# notebook: 02_postgis_connection.ipynb
from sqlalchemy import create_engine
import geopandas as gpd
import pandas as pd
from dotenv import load_dotenv
import os

load_dotenv()

# Conexi n segura usando variables de entorno
engine = create_engine(
    f"postgresql://{os.getenv('DB_USER')}:{os.getenv('DB_PASS')}"
    f"@{os.getenv('DB_HOST')}/{os.getenv('DB_NAME')}"
)

# Demo: Consulta espacial compleja
sql = """
WITH metro_buffer AS (
    SELECT ST_Buffer(geom::geography, 500)::geometry as buffer
    FROM estaciones_metro
)
SELECT
    p.id,
    p.direccion,
    p.precio,
    p.m2,
    p.precio / p.m2 as precio_m2,
    CASE
        WHEN EXISTS(SELECT 1 FROM metro_buffer mb
                    WHERE ST_Intersects(p.geom, mb.buffer))
        THEN 'Cerca de metro'
        ELSE 'Lejos de metro'
    END as acceso_metro
FROM propiedades p
WHERE p.comuna = 'Las Condes'
ORDER BY precio_m2 DESC
LIMIT 10
"""

df = gpd.read_postgis(sql, engine, geom_col='geom')
print(df[['direccion', 'precio_m2', 'acceso_metro']])
```

[00:25 - 00:30] APIs comerciales - Google Maps:

Tip

Enfatizar el costo de las APIs comerciales y la importancia del rate limiting.

```python
# notebook: 03_google_maps_api.ipynb
import googlemaps
import pandas as pd
from datetime import datetime
import time

# Cliente con API key
gmaps = googlemaps.Client(key=os.getenv('GOOGLE_API_KEY'))
```

```
9
10 # Demo: Geocodificaci n batch con manejo de errores
11 direcciones = [
12     "Av. Apoquindo 3000, Las Condes",
13     "Providencia 1234, Providencia",
14     "Estado 10, Santiago Centro"
15 ]
16
17 resultados = []
18 for direccion in direcciones:
19     try:
20         print(f"Geocodificando: {direccion}")
21         result = gmaps.geocode(f"{direccion}, Santiago, Chile")
22         if result:
23             location = result[0]['geometry']['location']
24             resultados.append({
25                 'direccion': direccion,
26                 'lat': location['lat'],
27                 'lon': location['lng'],
28                 'formatted': result[0]['formatted_address']
29             })
30         time.sleep(0.1)  # Rate limiting
31     except Exception as e:
32         print(f"Error en {direccion}: {e}")
33         resultados.append({'direccion': direccion, 'error': str(e)})
34
35 df_geocoded = pd.DataFrame(resultados)
36 print(df_geocoded)
```

4.5. Sección 3: Análisis Espacial Aplicado y Testing (25 minutos)

[00:30 - 00:35] Geocodificación masiva robusta:

Importante

Esta sección es crítica para proyectos con direcciones. Dedicar tiempo a explicar el manejo de errores.

```
1  # notebook: 04_geocoding_pipeline.ipynb
2  from geopy.geocoders import Nominatim
3  from geopy.extra.rate_limiter import RateLimiter
4  import pandas as pd
5  import geopandas as gpd
6  from shapely.geometry import Point
7
8  # Cargar dataset de prueba
9  df = pd.read_csv('data/raw/direcciones_propiedades.csv')
10 print(f"Total direcciones a geocodificar: {len(df)}")
11
12 # Configurar geocoder con rate limiting autom tico
13 geolocator = Nominatim(user_agent="clase_geoinformatica")
14 geocode = RateLimiter(geolocator.geocode, min_delay_seconds=1)
15
16 # Funci n robusta con fallback
17 def geocodificar_chile(direccion, comuna=None, retry=3):
18     """Geocodifica con m ltiples intentos y variaciones"""
19     attempts = [
20         f"{direccion}, {comuna}, Santiago, Chile" if comuna else None,
21         f"{direccion}, Santiago, Chile",
22         f"{direccion}, Chile",
23         direccion
```

```
    ]

    for attempt in filter(None, attempts):
        for i in range(retry):
            try:
                location = geocode(attempt)
                if location:
                    # Validar que est  en Chile
                    if -36 < location.latitude < -17 and -76 < location.
    longitude < -66:
                        return {
                            'lat': location.latitude,
                            'lon': location.longitude,
                            'confianza': 'alta' if comuna in attempt else 'media
    ',
                            'direccion_usada': attempt
                        }
            except Exception as e:
                print(f"Intento {i+1} fall : {e}")
                time.sleep(2 ** i)  # Exponential backoff

    return {'lat': None, 'lon': None, 'confianza': 'nula', 'error': 'No
    geocodificado'}

# Aplicar a subset para demo (en producci n usar Dask)
sample = df.head(20)
results = sample.apply(
    lambda row: geocodificar_chile(row['direccion'], row.get('comuna')),
    axis=1
)

# Crear GeoDataFrame con resultados
geocoded = pd.concat([sample, pd.DataFrame(list(results))], axis=1)
geocoded = geocoded[geocoded['lat'].notna()]

geometry = [Point(xy) for xy in zip(geocoded['lon'], geocoded['lat'])]
geo_df = gpd.GeoDataFrame(geocoded, geometry=geometry, crs='EPSG:4326')

# Estad sticas
print(f"Exitosas: {len(geocoded)}/{len(sample)} ({len(geocoded)/len(sample)
    *100:.1f}%)")
print(f"Confianza alta: {(geocoded['confianza']=='alta').sum()}")
print(f"Confianza media: {(geocoded['confianza']=='media').sum()}")
```

[00:35 - 00:42] Áreas de influencia reales con isócronas:

> Demo en vivo
>
> Esta demo es visualmente impactante. Mostrar la diferencia entre buffer circular e isócrona real.

```
# notebook: 05_isochrones.ipynb
import osmnx as ox
import networkx as nx
import geopandas as gpd
from shapely.geometry import Point, Polygon
import matplotlib.pyplot as plt

# Punto de inter s: Hospital Salvador
hospital_coords = (-70.6356, -33.4569)
hospital = Point(hospital_coords)

# Descargar red vial alrededor del hospital
```

```
G = ox.graph_from_point(hospital_coords, dist=2000, network_type='walk')
G = ox.project_graph(G)

# Encontrar nodo m s cercano
hospital_node = ox.distance.nearest_nodes(G, hospital_coords[0], hospital_coords
    [1])

# Calcular is cronas para 5, 10 y 15 minutos caminando
walk_speed = 4.5  # km/h
times = [5, 10, 15]  # minutos
isochrones = {}

for trip_time in times:
    # Distancia m xima en metros
    meters = walk_speed * 1000 / 60 * trip_time

    # Subgrafo alcanzable
    subgraph = nx.ego_graph(G, hospital_node, radius=meters, distance='length')

    # Extraer nodos del subgrafo
    node_points = [Point((data['x'], data['y']))
                   for node, data in subgraph.nodes(data=True)]

    # Crear pol gono convexo (simplificado)
    if len(node_points) >= 3:
        from shapely.ops import unary_union
        from shapely.geometry import MultiPoint
        isochrones[trip_time] = MultiPoint(node_points).convex_hull

# Visualizaci n comparativa
fig, axes = plt.subplots(1, 2, figsize=(15, 7))

# Buffer circular (m todo naive)
ax1 = axes[0]
for minutes in times:
    radius = walk_speed * 1000 / 60 * minutes
    circle = hospital.buffer(radius)
    gpd.GeoSeries([circle]).plot(ax=ax1, alpha=0.3,
                                 label=f'{minutes} min')
ax1.plot(*hospital.xy, 'r*', markersize=15, label='Hospital')
ax1.set_title('M todo Buffer Circular (Incorrecto)')
ax1.legend()

# Is crona real considerando calles
ax2 = axes[1]
colors = ['green', 'yellow', 'red']
for i, (minutes, geom) in enumerate(isochrones.items()):
    gpd.GeoSeries([geom]).plot(ax=ax2, alpha=0.3, color=colors[i],
                               label=f'{minutes} min')
ax2.plot(*hospital.xy, 'r*', markersize=15, label='Hospital')
ax2.set_title('Is crona Real (Correcto)')
ax2.legend()

plt.suptitle('Comparaci n: Buffer vs Is crona')
plt.tight_layout()
plt.show()

print("Observen c mo la is crona real sigue la red vial")
print("y no asume que se puede caminar a trav s de edificios")
```

[00:42 - 00:50] Clustering espacial para detectar patrones:

Tip

Relacionar con el proyecto de valoración inmobiliaria - detectar zonas de precios similares.

```python
# notebook: 06_spatial_clustering.ipynb
from sklearn.cluster import DBSCAN
import numpy as np
import geopandas as gpd
import matplotlib.pyplot as plt
from matplotlib.patches import Circle

# Cargar datos de propiedades
props = gpd.read_file('data/processed/propiedades_santiago.geojson')
props = props[props['precio'].notna()]

# Preparar features para clustering
# Incluir ubicaci n Y caracter sticas
coords = np.array([[p.x, p.y] for p in props.geometry])
precios_norm = props['precio'].values / props['precio'].std()
m2_norm = props['m2'].values / props['m2'].std()

# Feature matrix ponderada
X = np.column_stack([
    coords[:, 0] * 100,  # Peso alto a ubicaci n X
    coords[:, 1] * 100,  # Peso alto a ubicaci n Y
    precios_norm,        # Precio normalizado
    m2_norm              # Tama o normalizado
])

# DBSCAN - detectar clusters de propiedades similares
db = DBSCAN(eps=50, min_samples=5).fit(X)
props['cluster'] = db.labels_

# Estad sticas por cluster
print(f"Clusters encontrados: {len(set(db.labels_)) - (1 if -1 in db.labels_
    else 0)}")
print(f"Puntos ruido: {list(db.labels_).count(-1)}")

# An lisis por cluster
for cluster_id in set(db.labels_):
    if cluster_id != -1:  # Ignorar ruido
        cluster_data = props[props['cluster'] == cluster_id]
        print(f"\nCluster {cluster_id}:")
        print(f"  Propiedades: {len(cluster_data)}")
        print(f"  Precio promedio: ${cluster_data['precio'].mean():,.0f}")
        print(f"  M2 promedio: {cluster_data['m2'].mean():.1f}")
        print(f"  Precio/M2: ${(cluster_data['precio']/cluster_data['m2']).mean
    ():,.0f}")

# Visualizaci n
fig, ax = plt.subplots(figsize=(12, 10))

# Plot por clusters
for cluster_id in set(db.labels_):
    if cluster_id == -1:
        color = 'gray'
        label = 'Ruido'
    else:
        color = plt.cm.Set1(cluster_id)
        label = f'Cluster {cluster_id}'

    cluster_data = props[props['cluster'] == cluster_id]
```

```
    cluster_data.plot(ax=ax, color=color, label=label,
                      markersize=20, alpha=0.6)

ax.set_title('Clusters de Propiedades Similares')
ax.legend()
plt.show()

# Crear pol gonos de mercado para cada cluster
from shapely.ops import unary_union
market_zones = []
for cluster_id in set(db.labels_):
    if cluster_id != -1:
        cluster_data = props[props['cluster'] == cluster_id]
        zone = unary_union(cluster_data.geometry).convex_hull
        market_zones.append({
            'cluster': cluster_id,
            'geometry': zone,
            'precio_m2_promedio': (cluster_data['precio']/cluster_data['m2']).
    mean()
        })

zones_gdf = gpd.GeoDataFrame(market_zones)
zones_gdf.to_file('data/processed/zonas_mercado.geojson', driver='GeoJSON')
print("\nZonas de mercado exportadas a GeoJSON")
```

4.6. Sección 4: Optimización, Escalabilidad y Monitoreo (20 minutos)

[00:50 - 00:55] Testing de Componentes Geoespaciales:

Importante

Esta sección es crítica para la calidad del software geoespacial.

```
# tests/test_spatial_operations.py
import pytest
import geopandas as gpd
from shapely.geometry import Point, Polygon
import numpy as np

class TestSpatialValidation:
    """Tests para validaci n de geometr as"""

    def test_polygon_validity(self):
        """Verificar que detectamos pol gonos inv lidos"""
        # Pol gono con auto-intersecci n (inv lido)
        coords = [(0,0), (2,2), (2,0), (0,2), (0,0)]
        invalid_poly = Polygon(coords)
        assert not invalid_poly.is_valid

        # Reparar con buffer(0)
        fixed_poly = invalid_poly.buffer(0)
        assert fixed_poly.is_valid

    def test_crs_transformation(self):
        """Verificar transformaci n de coordenadas"""
        # Punto en Santiago (WGS84)
        point = Point(-70.65, -33.45)
        gdf = gpd.GeoDataFrame([1], geometry=[point], crs='EPSG:4326')

        # Transformar a UTM Zone 19S
        gdf_utm = gdf.to_crs('EPSG:32719')
```

```
        # Verificar que las coordenadas cambiaron
        assert gdf_utm.geometry[0].x != point.x
        assert abs(gdf_utm.geometry[0].x - 347000) < 1000  # Aprox

    def test_spatial_join_performance(self):
        """Test de rendimiento para spatial join"""
        import time

        # Crear datasets de prueba
        points = [Point(np.random.uniform(-71, -70),
                        np.random.uniform(-34, -33))
                  for _ in range(1000)]
        polygons = [Point(x, y).buffer(0.01)
                    for x in np.linspace(-71, -70, 10)
                    for y in np.linspace(-34, -33, 10)]

        gdf_points = gpd.GeoDataFrame(geometry=points, crs='EPSG:4326')
        gdf_polygons = gpd.GeoDataFrame(geometry=polygons, crs='EPSG:4326')

        # Medir tiempo con  ndice  espacial
        start = time.time()
        result = gpd.sjoin(gdf_points, gdf_polygons, predicate='within')
        time_with_index = time.time() - start

        assert time_with_index < 1.0  # Debe ser r pido
        assert len(result) > 0  # Debe haber matches

# Ejecutar con: pytest tests/ -v --cov=src
```

[00:55 - 01:00] Monitoreo y Observabilidad:

```
# src/monitoring/metrics.py
from prometheus_client import Counter, Histogram, Gauge
import time
import functools

# M tricas Prometheus
geo_requests = Counter('geo_requests_total',
                       'Total geocoding requests',
                       ['service', 'status'])
geo_latency = Histogram('geo_request_duration_seconds',
                        'Geocoding request latency')
active_connections = Gauge('postgis_connections_active',
                           'Active PostGIS connections')

def monitor_performance(func):
    """Decorator para monitorear funciones geoespaciales"""
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        start = time.time()
        try:
            result = func(*args, **kwargs)
            geo_requests.labels(service=func.__name__,
                                status='success').inc()
            return result
        except Exception as e:
            geo_requests.labels(service=func.__name__,
                                status='error').inc()
            raise e
        finally:
            geo_latency.observe(time.time() - start)
    return wrapper

```

```
@monitor_performance
def geocode_address(address):
    """Geocodificar con monitoreo autom tico"""
    # Tu c digo de geocodificaci n aqu
    pass
```

[01:00 - 01:05] Procesamiento eficiente con Dask:

> **Importante**
>
> Si los estudiantes no conocen Dask, explicar que es "Pandas paralelo para big data".

```
# notebook: 07_optimization.ipynb
import dask.dataframe as dd
import dask_geopandas as dgpd
import geopandas as gpd
import time

# Comparaci n: Pandas vs Dask
print("Dataset grande: 1 mill n de puntos")

# M todo tradicional (lento)
start = time.time()
gdf = gpd.read_file('data/raw/million_points.geojson')
gdf['buffer_100'] = gdf.buffer(100)
tradicional_time = time.time() - start
print(f"Tiempo con GeoPandas: {tradicional_time:.2f} segundos")

# M todo con Dask (r pido)
start = time.time()
ddf = dgpd.read_file('data/raw/million_points.geojson', npartitions=8)
ddf['buffer_100'] = ddf.geometry.buffer(100)
result = ddf.compute()  # Ejecutar en paralelo
dask_time = time.time() - start
print(f"Tiempo con Dask: {dask_time:.2f} segundos")
print(f"Speedup: {tradicional_time/dask_time:.2f}x m s r pido")

# Procesamiento por chunks para memoria limitada
def procesar_chunk(chunk_df):
    """Procesa un chunk de datos"""
    chunk_df['area'] = chunk_df.geometry.area
    chunk_df['perimetro'] = chunk_df.geometry.length
    chunk_df['compacidad'] = 4 * np.pi * chunk_df['area'] / chunk_df['perimetro']
    **2
    return chunk_df[chunk_df['compacidad'] > 0.7]  # Filtrar formas compactas

# Procesar archivo enorme sin cargar todo en memoria
chunks = []
for chunk in pd.read_csv('huge_file.csv', chunksize=10000):
    chunk['geometry'] = chunk.apply(lambda x: Point(x['lon'], x['lat']), axis=1)
    chunk_gdf = gpd.GeoDataFrame(chunk, crs='EPSG:4326')
    processed = procesar_chunk(chunk_gdf)
    chunks.append(processed)
    print(f"Procesado chunk con {len(processed)} registros")

result = pd.concat(chunks, ignore_index=True)
print(f"Total procesado: {len(result)} registros")
```

[01:05 - 01:10] Índices espaciales y caché:

```
# notebook: 08_spatial_index.ipynb
from rtree import index
import pickle
import time
```

```python

class SpatialCache:
    """Cache espacial con R-tree para b squedas ultra-r pidas"""

    def __init__(self):
        # Propiedades del  ndice  R-tree
        p = index.Property()
        p.dimension = 2
        p.variant = index.RT_Star  # Mejor algoritmo
        p.fill_factor = 0.7

        self.idx = index.Index(properties=p)
        self.cache = {}
        self.stats = {'hits': 0, 'misses': 0}

    def load_data(self, gdf):
        """Cargar GeoDataFrame al  ndice """
        print(f"Indexando {len(gdf)} features...")
        start = time.time()

        for idx, row in gdf.iterrows():
            bounds = row.geometry.bounds
            self.idx.insert(idx, bounds)
            self.cache[idx] = row.to_dict()

        print(f"Indexado en {time.time()-start:.2f} segundos")

    def query_bbox(self, bbox, use_cache=True):
        """B squeda por bounding box"""
        if use_cache:
            self.stats['hits'] += 1
            candidates = list(self.idx.intersection(bbox))
            return [self.cache[i] for i in candidates]
        else:
            self.stats['misses'] += 1
            # Sin  ndice  - b squeda lineal (lenta)
            results = []
            for key, item in self.cache.items():
                # Verificaci n manual (muy lenta)
                geom_bounds = item['geometry'].bounds
                if (bbox[0] <= geom_bounds[2] and bbox[2] >= geom_bounds[0] and
                    bbox[1] <= geom_bounds[3] and bbox[3] >= geom_bounds[1]):
                    results.append(item)
            return results

    def print_stats(self):
        print(f"Cache hits: {self.stats['hits']}")
        print(f"Cache misses: {self.stats['misses']}")
        hit_rate = self.stats['hits'] / (self.stats['hits'] + self.stats['misses
    ']) * 100
        print(f"Hit rate: {hit_rate:.1f}%")

# Demo de uso
cache = SpatialCache()
comunas = gpd.read_file('data/processed/comunas.geojson')
cache.load_data(comunas)

# B squeda con  ndice  (r pida)
bbox_santiago = [-70.7, -33.5, -70.5, -33.4]
start = time.time()
results_indexed = cache.query_bbox(bbox_santiago, use_cache=True)
time_indexed = time.time() - start

```

```
# B squeda sin  ndice  (lenta)
start = time.time()
results_linear = cache.query_bbox(bbox_santiago, use_cache=False)
time_linear = time.time() - start

print(f"\nResultados encontrados: {len(results_indexed)}")
print(f"Tiempo CON  ndice : {time_indexed*1000:.2f} ms")
print(f"Tiempo SIN  ndice : {time_linear*1000:.2f} ms")
print(f"Speedup: {time_linear/time_indexed:.1f}x m s r pido")

cache.print_stats()
```

4.7. Sección 5: Deployment con Arquitectura de Microservicios (20 minutos)

[01:00 - 01:07] API REST con FastAPI:

Demo en vivo

Ejecutar la API y probarla con Postman o curl en tiempo real.

```
# archivo: src/api/main.py
from fastapi import FastAPI, HTTPException, Query
from fastapi.middleware.cors import CORSMiddleware
from pydantic import BaseModel, validator
import geopandas as gpd
from shapely.geometry import Point
import json
from typing import Optional, List
from datetime import datetime
import redis
import hashlib

# Inicializar FastAPI
app = FastAPI(
    title="GeoAPI Inmobiliaria",
    description="API para an lisis geoespacial de propiedades",
    version="1.0.0"
)

# CORS para permitir frontend
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_methods=["*"],
    allow_headers=["*"],
)

# Cache Redis
cache = redis.Redis(host='localhost', port=6379, decode_responses=True)

# Cargar datos al iniciar
print("Cargando datos geoespaciales...")
comunas = gpd.read_file("data/processed/comunas.geojson")
propiedades = gpd.read_file("data/processed/propiedades.geojson")
print(f"Datos cargados: {len(comunas)} comunas, {len(propiedades)} propiedades")

# Modelos Pydantic para validaci n
class LocationRequest(BaseModel):
    lat: float
    lon: float

    @validator('lat')
```

```
    def validate_lat(cls, v):
        if not -90 <= v <= 90:
            raise ValueError('Latitud debe estar entre -90 y 90')
        return v

    @validator('lon')
    def validate_lon(cls, v):
        if not -180 <= v <= 180:
            raise ValueError('Longitud debe estar entre -180 y 180')
        return v

class PropertyFilter(BaseModel):
    precio_min: Optional[float] = 0
    precio_max: Optional[float] = 1e9
    m2_min: Optional[float] = 0
    m2_max: Optional[float] = 1000
    comuna: Optional[str] = None
    tipo: Optional[str] = None

# Endpoints
@app.get("/")
async def root():
    return {
        "message": "API Geoespacial Funcionando",
        "endpoints": [
            "/docs",
            "/api/comuna",
            "/api/propiedades/cercanas",
            "/api/analisis/precio-m2",
            "/api/isocronas"
        ]
    }

@app.post("/api/comuna")
async def get_comuna(location: LocationRequest):
    """Obtener informaci n de la comuna para una ubicaci n"""

    # Cache key
    cache_key = f"comuna:{location.lat}:{location.lon}"
    cached = cache.get(cache_key)
    if cached:
        return json.loads(cached)

    point = Point(location.lon, location.lat)

    for idx, comuna in comunas.iterrows():
        if comuna.geometry.contains(point):
            result = {
                "comuna": comuna['nombre'],
                "region": comuna['region'],
                "poblacion": int(comuna['poblacion']),
                "area_km2": comuna.geometry.area / 1e6,
                "timestamp": datetime.now().isoformat()
            }

            # Guardar en cache por 1 hora
            cache.setex(cache_key, 3600, json.dumps(result))
            return result

    raise HTTPException(status_code=404, detail="Ubicaci n fuera de Chile")

@app.get("/api/propiedades/cercanas")
async def propiedades_cercanas(
```

```
    lat: float = Query(..., description="Latitud"),
    lon: float = Query(..., description="Longitud"),
    radio: float = Query(500, description="Radio en metros"),
    limit: int = Query(10, description="M ximo de resultados")
):
    """Encontrar propiedades dentro de un radio"""

    point = Point(lon, lat)
    point_utm = gpd.GeoSeries([point], crs='EPSG:4326').to_crs('EPSG:32719')[0]

    # Buffer en metros
    buffer = point_utm.buffer(radio)

    # Filtrar propiedades
    props_utm = propiedades.to_crs('EPSG:32719')
    mask = props_utm.geometry.within(buffer)
    cercanas = propiedades[mask].copy()

    if len(cercanas) == 0:
        return {"message": "No hay propiedades en el radio especificado", "
    results": []}

    # Calcular distancias
    cercanas['distancia'] = props_utm[mask].geometry.distance(point_utm)
    cercanas = cercanas.nsmallest(limit, 'distancia')

    # Preparar respuesta
    results = []
    for idx, prop in cercanas.iterrows():
        results.append({
            "id": int(idx),
            "direccion": prop.get('direccion', 'Sin direcci n'),
            "precio": float(prop['precio']),
            "m2": float(prop['m2']),
            "precio_m2": float(prop['precio'] / prop['m2']),
            "distancia": float(prop['distancia']),
            "lat": prop.geometry.y,
            "lon": prop.geometry.x
        })

    return {
        "centro": {"lat": lat, "lon": lon},
        "radio": radio,
        "total": len(results),
        "results": results
    }

@app.post("/api/analisis/precio-m2")
async def analizar_precio_m2(filters: PropertyFilter):
    """An lisis estad stico de precios por m2"""

    # Aplicar filtros
    filtered = propiedades.copy()
    filtered = filtered[
        (filtered['precio'] >= filters.precio_min) &
        (filtered['precio'] <= filters.precio_max) &
        (filtered['m2'] >= filters.m2_min) &
        (filtered['m2'] <= filters.m2_max)
    ]

    if filters.comuna:
        filtered = filtered[filtered['comuna'] == filters.comuna]

```

```
    if filters.tipo:
        filtered = filtered[filtered['tipo'] == filters.tipo]

    if len(filtered) == 0:
        raise HTTPException(status_code=404, detail="No hay propiedades con esos
    filtros")

    # Calcular estad sticas
    filtered['precio_m2'] = filtered['precio'] / filtered['m2']

    return {
        "total_propiedades": len(filtered),
        "estadisticas": {
            "precio_m2_promedio": float(filtered['precio_m2'].mean()),
            "precio_m2_mediana": float(filtered['precio_m2'].median()),
            "precio_m2_min": float(filtered['precio_m2'].min()),
            "precio_m2_max": float(filtered['precio_m2'].max()),
            "precio_m2_std": float(filtered['precio_m2'].std())
        },
        "distribucion": {
            "q25": float(filtered['precio_m2'].quantile(0.25)),
            "q50": float(filtered['precio_m2'].quantile(0.50)),
            "q75": float(filtered['precio_m2'].quantile(0.75)),
            "q90": float(filtered['precio_m2'].quantile(0.90))
        },
        "filtros_aplicados": filters.dict()
    }

# Ejecutar con:
# uvicorn main:app --reload --host 0.0.0.0 --port 8000
```

[01:07 - 01:15] Dashboard interactivo con Streamlit:

> **Tip**
>
> Streamlit es perfecto para prototipos rápidos. Mostrar cómo en 50 líneas tienen un dashboard.

```
# archivo: src/visualization/dashboard.py
import streamlit as st
import geopandas as gpd
import pandas as pd
import folium
from streamlit_folium import st_folium
import plotly.express as px
import requests

st.set_page_config(
    page_title="Dashboard Inmobiliario",
    page_icon="    ",
    layout="wide"
)

# CSS personalizado
st.markdown("""
<style>
.big-font {
    font-size:20px !important;
    font-weight: bold;
}
</style>
""", unsafe_allow_html=True)

```

```
# T tulo
st.title("     Dashboard de An lisis Inmobiliario")
st.markdown("### An lisis geoespacial de propiedades en Santiago")

# Sidebar
with st.sidebar:
    st.header("       Filtros")

    # Filtros
    comuna = st.selectbox(
        "Comuna",
        ["Todas", "Las Condes", "Providencia", "Vitacura", "Santiago Centro"]
    )

    precio_range = st.slider(
        "Rango de precio (UF)",
        min_value=1000,
        max_value=50000,
        value=(5000, 15000),
        step=500,
        format="%d UF"
    )

    m2_range = st.slider(
        "Superficie (m  )",
        min_value=30,
        max_value=500,
        value=(50, 200),
        step=10
    )

    tipo_propiedad = st.multiselect(
        "Tipo de propiedad",
        ["Departamento", "Casa", "Oficina"],
        default=["Departamento", "Casa"]
    )

    st.markdown("---")

    # An lisis
    if st.button("     Actualizar An lisis"):
        st.experimental_rerun()

# Layout principal
col1, col2, col3, col4 = st.columns(4)

# KPIs
@st.cache_data
def load_data():
    """Cargar y cachear datos"""
    props = gpd.read_file('data/processed/propiedades.geojson')
    return props

props = load_data()

# Aplicar filtros
if comuna != "Todas":
    props = props[props['comuna'] == comuna]

props = props[
    (props['precio_uf'] >= precio_range[0]) &
    (props['precio_uf'] <= precio_range[1]) &
    (props['m2'] >= m2_range[0]) &
```

```
    (props['m2'] <= m2_range[1]) &
    (props['tipo'].isin(tipo_propiedad))
]

# M tricas
with col1:
    st.metric(
        "Total Propiedades",
        f"{len(props):,}",
        f"{len(props) - 100:+,} vs mes anterior"
    )

with col2:
    precio_promedio = props['precio_uf'].mean()
    st.metric(
        "Precio Promedio",
        f"{precio_promedio:,.0f} UF",
        "+2.3% vs mes anterior"
    )

with col3:
    m2_promedio = props['m2'].mean()
    st.metric(
        "M   Promedio",
        f"{m2_promedio:.0f} m  ",
        "-1.2% vs mes anterior"
    )

with col4:
    precio_m2 = (props['precio_uf'] / props['m2']).mean()
    st.metric(
        "UF/M   Promedio",
        f"{precio_m2:.1f}",
        "+3.1% vs mes anterior"
    )

st.markdown("---")

# Mapa y gr ficos
col_mapa, col_graficos = st.columns([2, 1])

with col_mapa:
    st.markdown('<p class="big-font">     Mapa de Propiedades</p>',
                unsafe_allow_html=True)

    # Crear mapa
    m = folium.Map(location=[-33.45, -70.65], zoom_start=11)

    # Agregar marcadores con clustering
    from folium.plugins import MarkerCluster
    marker_cluster = MarkerCluster().add_to(m)

    for idx, row in props.iterrows():
        folium.Marker(
            [row.geometry.y, row.geometry.x],
            popup=f"""
            <b>{row['tipo']}</b><br>
            Precio: {row['precio_uf']:,.0f} UF<br>
            Superficie: {row['m2']:.0f} m  <br>
            Comuna: {row['comuna']}
            """,
            icon=folium.Icon(
                color='green' if row['precio_uf'] < precio_promedio else 'red',
```

```
                icon='home'
            )
        ).add_to(marker_cluster)

    # Mostrar mapa
    st_folium(m, height=400, width=None)

with col_graficos:
    st.markdown('<p class="big-font">     An lisis </p>',
                unsafe_allow_html=True)

    # Gr fico 1: Distribuci n de precios
    fig1 = px.histogram(
        props,
        x='precio_uf',
        nbins=30,
        title="Distribuci n de Precios",
        labels={'precio_uf': 'Precio (UF)', 'count': 'Cantidad'}
    )
    fig1.update_layout(height=200)
    st.plotly_chart(fig1, use_container_width=True)

    # Gr fico 2: Precio por comuna
    precio_comuna = props.groupby('comuna')['precio_uf'].mean().sort_values()
    fig2 = px.bar(
        x=precio_comuna.values,
        y=precio_comuna.index,
        orientation='h',
        title="Precio Promedio por Comuna",
        labels={'x': 'Precio (UF)', 'y': 'Comuna'}
    )
    fig2.update_layout(height=200)
    st.plotly_chart(fig2, use_container_width=True)

st.markdown("---")

# Tabla detallada
st.markdown('<p class="big-font">     Detalle de Propiedades</p>',
            unsafe_allow_html=True)

# Preparar datos para tabla
tabla = props[['tipo', 'comuna', 'precio_uf', 'm2', 'dormitorios', 'banos']].
    copy()
tabla['precio_m2'] = tabla['precio_uf'] / tabla['m2']
tabla = tabla.round(1)

# Mostrar tabla con formato
st.dataframe(
    tabla,
    use_container_width=True,
    hide_index=True,
    column_config={
        "precio_uf": st.column_config.NumberColumn(
            "Precio (UF)",
            format="%,.0f UF"
        ),
        "m2": st.column_config.NumberColumn(
            "Superficie",
            format="%d  m  "
        ),
        "precio_m2": st.column_config.NumberColumn(
            "UF/ m  ",
            format="%.1f"
```

```
        )
    }
)

# Footer
st.markdown("---")
st.caption("Dashboard actualizado en tiempo real | Datos: Portal Inmobiliario")

# Ejecutar con:
# streamlit run dashboard.py
```

4.8. Sección 6: Desafíos Comunes y Soluciones (10 minutos)

[01:15 - 01:20] Resolución de Problemas Comunes:

Demo en vivo

Mostrar casos reales de errores y cómo solucionarlos.

```
# Problema 1: Mezcla de CRS
def safe_spatial_join(gdf1, gdf2, **kwargs):
    """Spatial join con validaci n de CRS"""
    if gdf1.crs != gdf2.crs:
        print(f"CRS mismatch: {gdf1.crs} vs {gdf2.crs}")
        print("Transformando al CRS del primer GeoDataFrame...")
        gdf2 = gdf2.to_crs(gdf1.crs)

    return gpd.sjoin(gdf1, gdf2, **kwargs)

# Problema 2: Geometr as inv lidas
def clean_geometries(gdf):
    """Limpiar y reparar geometr as problem ticas"""
    # Detectar geometr as inv lidas
    invalid_mask = ~gdf.geometry.is_valid
    print(f"Geometr as inv lidas: {invalid_mask.sum()}")

    if invalid_mask.any():
        # Intentar reparar con buffer(0)
        gdf.loc[invalid_mask, 'geometry'] = \
            gdf.loc[invalid_mask, 'geometry'].buffer(0)

        # Verificar de nuevo
        still_invalid = ~gdf.geometry.is_valid
        if still_invalid.any():
            print(f"No se pudieron reparar {still_invalid.sum()} geometr as")
            # Opci n: eliminar o usar convex hull
            gdf.loc[still_invalid, 'geometry'] = \
                gdf.loc[still_invalid, 'geometry'].convex_hull

    return gdf

# Problema 3: Datos fuera de memoria
def process_large_file(filepath, chunksize=10000):
    """Procesar archivo grande por chunks"""
    results = []

    for chunk_df in pd.read_csv(filepath, chunksize=chunksize):
        # Convertir a GeoDataFrame
        geometry = [Point(xy) for xy in zip(chunk_df.lon, chunk_df.lat)]
        chunk_gdf = gpd.GeoDataFrame(chunk_df, geometry=geometry)

        # Procesar chunk
```

```
        processed = your_processing_function(chunk_gdf)
        results.append(processed)

        print(f"Procesado chunk con {len(chunk_gdf)} registros")

    return pd.concat(results, ignore_index=True)

# Problema 4: Rate limiting en APIs
from tenacity import retry, stop_after_attempt, wait_exponential

@retry(stop=stop_after_attempt(3),
       wait=wait_exponential(multiplier=1, min=4, max=10))
def geocode_with_retry(address):
    """Geocodificar con reintentos y backoff exponencial"""
    try:
        result = geocoder.geocode(address)
        return result
    except RateLimitError:
        print("Rate limit alcanzado, esperando...")
        raise
    except Exception as e:
        print(f"Error geocodificando {address}: {e}")
        raise
```

Matriz de Decisión Tecnológica:

Criterio	PostGIS	MongoDB	Elasticsearch	BigQuery
Consultas espaciales				
Escalabilidad				
ACID				
Costo				
Tiempo real				
Análisis complejo				

Cuadro 1: Comparación de tecnologías para almacenamiento geoespacial

4.9. Cierre y Síntesis (10 minutos)

[01:20 - 01:25] Demo integrada del pipeline completo:

Demo en vivo

Mostrar el flujo completo: desde dato crudo hasta dashboard en producción.

```
# Demo del pipeline completo

# 1. Obtener datos
python src/etl/download_osm.py --place "Santiago" --tipo "hospitals"

# 2. Procesar y limpiar
python src/etl/process_data.py --input raw/hospitals.json --output processed/

# 3. An lisis espacial
python src/analysis/clustering.py --data processed/hospitals.geojson

# 4. Levantar API
uvicorn src.api.main:app --reload &

# 5. Levantar dashboard
streamlit run src/visualization/dashboard.py &

```

```
# 6. Probar API
curl http://localhost:8000/api/comuna -X POST \
  -H "Content-Type: application/json" \
  -d '{"lat": -33.45, "lon": -70.65}'

echo "Pipeline completo funcionando!"
```

[01:25 - 01:28] Mejores prácticas y errores comunes:

> **Importante**
>
> Enfatizar estos puntos - son los errores más comunes en proyectos reales.

- **Error #1**: No validar CRS

  "SIEMPRE verifiquen el CRS. Mezclar WGS84 con UTM es el error más común."

- **Error #2**: No usar índices espaciales

  "Sin índices, una consulta de 1 segundo puede tomar 1 minuto."

- **Error #3**: Cargar todo en memoria

  Üsen chunks o Dask. No intenten cargar 1GB de GeoJSON en pandas."

- **Error #4**: No cachear resultados costosos

  "Geocodificar la misma dirección 100 veces es desperdiciar dinero y tiempo."

[01:28 - 01:30] Q&A y recursos:

> **Tip**
>
> Dejar tiempo para preguntas específicas de sus proyectos.

Recursos esenciales para sus proyectos:

- **Datos Chile**: IDE.cl, datos.gob.cl, geoportal.cl
- **Documentación**: geopandas.org, postgis.net
- **Comunidad**: GIS StackExchange, r/gis
- **Mi email**: fparra@usach.cl para dudas específicas

5. Material de Apoyo

5.1. Troubleshooting Común

Problema	Solución

ImportError: No module named 'gdal'	`conda install -c conda-forge gdal`
	`# o`
	`sudo apt-get install gdal-bin python3-gdal`

CRS mismatch warning	`# Siempre transformar al mismo CRS` `gdf1 = gdf1.to_crs(gdf2.crs)`

Memory error con archivo grande	`# Usar chunks o Dask`
	`for chunk in pd.read_csv('big.csv', chunksize=10000):`
	`    process(chunk)`

Geocoding rate limit	`from geopy.extra.rate_limiter import RateLimiter`
	`geocode = RateLimiter(geolocator.geocode,`
	`                      min_delay_seconds=1)`

PostGIS connection refused	`# Verificar que PostgreSQL está corriendo`
	`sudo service postgresql status`
	`# Verificar puerto y host en connection string`

5.2. Configuración de Ambiente

5.2.1. Docker Compose para desarrollo

```yaml
# docker-compose.yml
version: '3.8'

services:
  postgis:
    image: postgis/postgis:14-3.2
    container_name: postgis_dev
    environment:
      POSTGRES_DB: geodata
      POSTGRES_USER: geouser
      POSTGRES_PASSWORD: geopass123
      PGDATA: /var/lib/postgresql/data/pgdata
    ports:
      - "5432:5432"
    volumes:
      - pgdata:/var/lib/postgresql/data
      - ./init.sql:/docker-entrypoint-initdb.d/init.sql
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U geouser"]
      interval: 10s
      timeout: 5s
      retries: 5

  redis:
    image: redis:7-alpine
    container_name: redis_cache
    ports:
      - "6379:6379"
    command: redis-server --appendonly yes
```

```
    volumes:
      - redis_data:/data

  jupyter:
    build:
      context: .
      dockerfile: Dockerfile.jupyter
    container_name: jupyter_geo
    ports:
      - "8888:8888"
    volumes:
      - ./notebooks:/home/jovyan/work
      - ./data:/home/jovyan/data
    environment:
      JUPYTER_ENABLE_LAB: "yes"
      GRANT_SUDO: "yes"
    depends_on:
      - postgis

volumes:
  pgdata:
  redis_data:
```

5.2.2. Script de inicialización de base de datos

```
-- init.sql
CREATE EXTENSION IF NOT EXISTS postgis;
CREATE EXTENSION IF NOT EXISTS postgis_topology;
CREATE EXTENSION IF NOT EXISTS fuzzystrmatch;
CREATE EXTENSION IF NOT EXISTS postgis_tiger_geocoder;

-- Tabla de comunas
CREATE TABLE IF NOT EXISTS comunas (
    id SERIAL PRIMARY KEY,
    nombre VARCHAR(100) NOT NULL,
    region VARCHAR(100),
    provincia VARCHAR(100),
    poblacion INTEGER,
    superficie DECIMAL(10,2),
    geom GEOMETRY(Polygon, 4326)
);

--  ndices
CREATE INDEX idx_comunas_geom ON comunas USING GIST(geom);
CREATE INDEX idx_comunas_nombre ON comunas(nombre);

-- Tabla de propiedades
CREATE TABLE IF NOT EXISTS propiedades (
    id SERIAL PRIMARY KEY,
    direccion VARCHAR(255),
    comuna_id INTEGER REFERENCES comunas(id),
    tipo VARCHAR(50),
    precio DECIMAL(12,2),
    precio_uf DECIMAL(10,2),
    m2 DECIMAL(8,2),
    dormitorios INTEGER,
    banos INTEGER,
    estacionamientos INTEGER,
    fecha_publicacion DATE,
    geom GEOMETRY(Point, 4326)
);

```

```sql
--  ndices  para propiedades
CREATE INDEX idx_propiedades_geom ON propiedades USING GIST(geom);
CREATE INDEX idx_propiedades_precio ON propiedades(precio);
CREATE INDEX idx_propiedades_comuna ON propiedades(comuna_id);
CREATE INDEX idx_propiedades_tipo ON propiedades(tipo);

-- Vista materializada para estad sticas por comuna
CREATE MATERIALIZED VIEW mv_stats_comuna AS
SELECT
    c.id,
    c.nombre,
    COUNT(p.id) as total_propiedades,
    AVG(p.precio_uf) as precio_promedio_uf,
    AVG(p.m2) as m2_promedio,
    AVG(p.precio_uf / NULLIF(p.m2, 0)) as precio_m2_promedio,
    MIN(p.precio_uf) as precio_minimo,
    MAX(p.precio_uf) as precio_maximo,
    STDDEV(p.precio_uf) as precio_stddev
FROM comunas c
LEFT JOIN propiedades p ON c.id = p.comuna_id
GROUP BY c.id, c.nombre;

CREATE INDEX idx_mv_stats_comuna ON mv_stats_comuna(nombre);

-- Funci n para encontrar propiedades cercanas
CREATE OR REPLACE FUNCTION propiedades_cercanas(
    lat FLOAT,
    lon FLOAT,
    radio_metros FLOAT,
    limite INTEGER DEFAULT 10
)
RETURNS TABLE (
    id INTEGER,
    direccion VARCHAR,
    precio DECIMAL,
    distancia FLOAT
) AS $$
BEGIN
    RETURN QUERY
    SELECT
        p.id,
        p.direccion,
        p.precio,
        ST_Distance(
            p.geom::geography,
            ST_SetSRID(ST_MakePoint(lon, lat), 4326)::geography
        ) as distancia
    FROM propiedades p
    WHERE ST_DWithin(
        p.geom::geography,
        ST_SetSRID(ST_MakePoint(lon, lat), 4326)::geography,
        radio_metros
    )
    ORDER BY distancia
    LIMIT limite;
END;
$$ LANGUAGE plpgsql;

GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA public TO geouser;
GRANT ALL PRIVILEGES ON ALL SEQUENCES IN SCHEMA public TO geouser;
```

5.3. Ejercicios para Estudiantes

5.3.1. Ejercicio 1: Pipeline ETL Completo

Implementar un pipeline que:

1. Descargue datos de hospitales desde OSM
2. Calcule áreas de servicio (isócronas de 10 min)
3. Identifique zonas sin cobertura
4. Exporte resultados a PostGIS
5. Cree visualización en mapa web

5.3.2. Ejercicio 2: API de Geocodificación con Caché

Crear una API que:

1. Reciba direcciones en formato chileno
2. Geocodifique usando múltiples servicios (fallback)
3. Implemente caché Redis con TTL
4. Valide resultados dentro de Chile
5. Retorne GeoJSON válido

5.3.3. Ejercicio 3: Dashboard de Análisis Inmobiliario

Desarrollar dashboard que muestre:

1. Mapa de calor de precios
2. Clustering de propiedades similares
3. Predicción de precios con features espaciales
4. Comparación entre comunas
5. Exportación de reportes PDF

6. Evaluación y Rúbrica

6.1. Participación en Clase

- **Excelente (7.0)**: Implementa demos, hace preguntas relevantes
- **Bueno (5.5)**: Sigue las demos, participa ocasionalmente
- **Regular (4.0)**: Presente pero pasivo
- **Insuficiente (¡4.0)**: No participa o ausente

6.2. Proyecto Pipeline (Tarea)

Criterio	Descripción	Ponderación
Estructura	Proyecto bien organizado con carpetas apropiadas	15 %
ETL	Pipeline funcional de extracción y transformación	20 %
Análisis	Implementa al menos 2 análisis espaciales complejos	25 %
Optimización	Usa índices, caché o procesamiento paralelo	15 %
API/Dashboard	Endpoint REST o visualización interactiva funcionando	20 %
Documentación	README claro con instrucciones de instalación	5 %

7. Notas Post-Clase

7.1. Seguimiento

- Crear canal Slack/Discord para dudas de implementación
- Compartir repositorio con código de los ejemplos
- Office hours para ayuda con proyectos específicos
- Grabar demos para estudiantes que falten

7.2. Material Adicional

- Video: "Building Production GeoAPIs PyCon 2024
- Tutorial: "PostGIS Performance Tips Paul Ramsey
- Curso: "Scalable Geospatial Analytics Coursera
- Libro: "Geospatial Development By Example with Python"

7.3. Preparación Próxima Clase

La Clase 5 cubrirá .Análisis espacial y geoestadística". Los estudiantes deberían:

- Tener su pipeline básico funcionando
- Haber identificado datos para su proyecto
- Revisar conceptos de estadística espacial
- Instalar librerías: PySAL, scikit-gstat